mathematics

MDPI

Article

Mining High-Efficiency Itemsets with Negative Utilities

Irfan Yildirim

Department of Computer Engineering, Faculty of Engineering and Architecture, Erzurum Technical University, 25050 Erzurum, Türkiye; irfan.yildirim@erzurum.edu.tr

Abstract: High-efficiency itemset mining has recently emerged as a new problem in itemset mining. An itemset is classified as a high-efficiency itemset if its utility-to-investment ratio meets or exceeds a specified efficiency threshold. The goal is to discover all high-efficiency itemsets in a given database. However, solving the problem is computationally complex, due to the large search space involved. To effectively address this problem, several algorithms have been proposed that assume that databases contain only positive utilities. However, real-world databases often contain negative utilities. When the existing algorithms are applied to such databases, they fail to discover the complete set of itemsets, due to their limitations in handling negative utilities. This study proposes a novel algorithm, MHEINU (mining high-efficiency itemset with negative utilities), designed to correctly mine a complete set of high-efficiency itemsets from databases that also contain negative utilities. MHEINU introduces two upper-bounds to efficiently and safely reduce the search space. Additionally, it features a list-based data structure to streamline the mining process and minimize costly database scans. Experimental results on various datasets containing negative utilities showed that MHEINU effectively discovered the complete set of high-efficiency itemsets, performing well in terms of runtime, number of join operations, and memory usage. Additionally, MHEINU demonstrated good scalability, making it suitable for large-scale datasets.

Keywords: data mining; efficiency; high-efficient itemset; pruning strategy; upper-bound

MSC: 68W99



Academic Editors: Huawen Liu and Raymond Lee

Received: 3 January 2025
Revised: 26 January 2025
Accepted: 6 February 2025
Published: 17 February 2025

Citation: Yildirim, I. Mining High-Efficiency Itemsets with Negative Utilities. *Mathematics* **2025**, *13*, 659. https://doi.org/10.3390/math13040659

Copyright: © 2025 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Frequent itemset mining (FIM) [1–7] is a popular data mining technique for identifying frequently occurring itemsets in transactional databases. However, FIM assumes that all items have equal importance and that each item may appear only once per transaction, which does not reflect real-world scenarios, where items have varying significance and quantities. For example, in retail, frequently purchased items may yield a low profit, while less frequent ones could be more profitable. As FIM focuses solely on item frequency, it may overlook itemsets that are less common but more valuable, making its results less meaningful or inadequate for decision-making processes.

To overcome this limitation, the problem of high-utility itemset mining (HUIM) [8] has been introduced. Unlike FIM, which targets frequent itemsets, HUIM identifies itemsets with high utility, considering factors like utility, profit, importance, weight, or other user-defined metrics. HUIM typically considers the internal utility (e.g., quantity sold) of items in transactions and the external utility (e.g., profit per unit) of items. The utility of an item in a transaction is derived by multiplying its external utility by its internal utility in that transaction. The utility of an itemset in a database is defined as the sum of the utilities of all

items within the itemset across the transactions in which the itemset appears. The HUIM problem has attracted considerable attention from researchers, leading to the development of various algorithms [9–22] to effectively solve the problem.

However, while HUIM addresses the shortcomings of FIM by identifying high-utility itemsets, it overlooks another crucial factor: investment, which involves allocating resources to achieve potential future benefits [23,24]. Thus, HUIM may not fully support decision-makers aiming to maximize profits, because it does not account for the capital required to acquire products before selling them [23]. For example, consider two itemsets, X (a smartphone and a smartwatch) and Y (a monitor and a keyboard), with utilities of \$400 and \$100, respectively. Based on these figures alone, HUIM suggests that X is more significant than Y for making profits. However, if the investments in X and Y are \$1000 and \$200, respectively, the utility-to-investment ratio for X is 0.4 (400/1000), while for Y it is 0.5 (100/200). This means that when planning to maximize future profits, it is more advantageous to invest in Y rather than X. This is because, with the same capital of \$100,000, a businessman can buy 100 units of X and earn a profit of \$40,000. However, if the businessman chooses to invest in Y instead, he would be able to buy 500 units and earn a profit of \$50,000. This demonstrates that a higher investment-to-utility ratio can provide better returns, even when the total utility of an itemset is lower [24].

To address this limitation of HUIM, the problem of high-efficiency itemset mining (HEIM) [23] was recently introduced, which considers both utility and investment. The efficiency of an itemset is calculated by dividing its utility by its investment. If this efficiency meets or exceeds a given minimum efficiency threshold ($minE$), the itemset is classified as a high-efficiency itemset (HEI). The goal of HEIM is to discover the complete and correct set of $HEIs$ for a given $minE$ within a dataset. However, solving the HEIM problem is a computationally complex task because of its large search space. Moreover, the efficiency values of itemsets cannot be used to prune the search space because the efficiency metric lacks monotonic or anti-monotonic properties [24]. In other words, for any itemset X and its superset X', the efficiency of X is not guaranteed to be less than or equal to the efficiency of X', which means the efficiency metric does not exhibit the monotonic property. Similarly, the efficiency of X is not guaranteed to be greater than or equal to the efficiency of X', meaning it does not follow the anti-monotonic property.

HEPM [23], the first algorithm proposed for the HEIM problem, introduced an efficiency upper-bound (eub) to overcome this challenge. This upper-bound guarantees the anti-monotonic property by overestimating the efficiency values of the itemsets and helps to prune the search space. Although eub is useful for this purpose, HEPM generates many candidates and requires multiple database scans [23]. To further improve the mining process for the HEIM problem, the HEPMiner [23] and MHEI [24] algorithms were developed. HEPMiner uses a list-based data structure, while MHEI employs a database projection and merging method. Both use their own upper-bounds, along with eub for further pruning of the search space. Among them, MHEI has demonstrated the best performance [24].

However, as noted in various studies [25,26], real-world databases often include items with negative utilities. For instance, large supermarket chains frequently run bundled or cross-promotion campaigns where some items are sold at a loss (negative utility to boost overall profits by increasing sales of related products) [25,27]. This strategy is not confined to retail but is widely observed across other sectors [28]. In the telecommunications industry, companies often offer discounted or bundled services that initially result in a loss but are designed to attract new customers or retain existing ones. For instance, discounted mobile phone plans bundled with data packages are commonly used to secure long-term profitability through upselling or cross-selling additional services. Similarly, in the hospitality sector, negative utility frequently occurs in bundled offerings. Hotels, for

example, may offer discounted accommodation rates paired with ancillary services such as dining, spa, or recreational activity packages. While the accommodation itself may incur a loss, the overall revenue is enhanced through the additional services. The software industry adopts similar strategies by offering heavily discounted subscription packages for bundled services. Although these bundles may generate an immediate loss (negative utility), they help foster customer retention and generate a foundation for recurring revenue. Moreover, negative-utility-based mining extends its applicability beyond retail areas and cross-selling strategies to various domains, including website clickstream data, biomedical applications, and mobile commerce [29,30]. Additionally, in the investment sector, particularly in stock portfolio management, negative utility can be leveraged to optimize returns by accounting for the daily losses of stocks. For example, in [31], a data mining framework was developed to help investors plan and manage diversified stock portfolios for long-term investments. By analyzing historical stock data and incorporating negative utility, the designed approach enables the optimization of portfolio strategies, leading to improved overall performance.

On the other hand, the existing HEIM algorithms were developed based on the assumption that the datasets contain only positive utilities. Therefore, when applied to such databases, they do not guarantee the discovery of the complete set of *HEIs*. The reason for this is that the existing algorithms prune the search space based on upper-bounds, which overestimate the efficiency of itemsets by considering all the items as having positive utilities. When a database contains items with negative utilities, these overestimations can turn into underestimations, causing incorrect pruning of itemsets that are actually *HEIs*. As a result, since the existing HEIM algorithms fail to discover the complete set of *HEIs* for datasets with negative utilities, they are insufficient for providing decision-makers with the necessary information. Therefore, it is crucial to develop methods that can effectively address this issue.

This study addresses a significant gap in existing HEIM algorithms, which fail to fully discover *HEIs* from datasets with negative utilities. To tackle this challenge, the study developed methods and techniques for the efficient and complete discovery of *HEIs* from such databases. The key contributions of this paper are as follows:

- Two novel upper-bounds, along with pruning strategies, are introduced to efficiently and safely prune the search space. These upper-bounds are designed to ensure that *HEIs* are not mistakenly pruned in the presence of negative utilities and to increase mining efficiency by early detection of itemsets that are not *HEIs*.
- A list-based data structure, *ELNU* (efficiency-list with negative utilities), is proposed to store essential information for mining *HEIs* from databases with negative utilities. This structure minimizes the need for costly database scans, making the mining process more efficient.
- An algorithm named MHEINU (mining high-efficiency itemset with negative utilities) is proposed to extract a correct and complete set of *HEIs* from databases with negative utilities, under a user-defined minimum efficiency threshold.
- Comprehensive experiments were conducted on a variety of datasets with differing characteristics. The results validated the efficiency and effectiveness of the MHEINU algorithm for mining *HEIs* in databases with negative utilities, evaluating factors such as runtime, number of join operations, memory consumption, scalability, and the number of discovered itemsets.

The rest of the paper is structured as follows. Section 2 covers the related work. Section 3 outlines the fundamental concepts of the HEIM problem and discusses the problem faced by existing methods when dealing with databases containing negative utilities. Section 4 introduces the upper-bounds and the data structure, and explains the mining procedure of the proposed MHEINU algorithm. Section 5 provides the experimental

results. Section 6 discusses some limitations of the HEIM problem. Finally, Section 7 concludes the paper and suggests directions for future research.

2. Related Work

High-utility itemset mining (HUIM) [8] aims to discover itemsets with high utility values in databases. The development of various algorithms has enhanced the efficiency of solving HUIM, starting with the two-phase algorithm [8]. This algorithm generates candidate itemsets in the first phase and identifies high-utility itemsets in the second phase, requiring multiple database scans, making it time-consuming. Subsequent algorithms, such as UP-Growth and UP-Growth+ [9], were introduced to reduce the number of database scans and candidates generated by utilizing their own techniques. However, they still struggle with the computational complexity of candidate generation processes. To address the challenges of candidate generation, single-phase algorithms like HUI-Miner [10], FHM [11], d2HUP [12], HUP-Miner [13], EFIM [14], IMHUP [15], mHUIMiner [16], HMiner [17], UPB-Miner [18], iMEFIM [19], and Hamm[20] were developed. These algorithms minimize database scans using tailored data structures or techniques such as database projection and merging. They significantly improve the HUIM process by reducing the search space with effective pruning strategies. Beyond classical algorithms, specialized methods have been developed for various HUIM extensions to address real-world application needs, such as mining closed *HUIs* [32], sequential *HUIs* [33], top-k *HUIs* [34,35], correlated *HUIs* [36], itemsets ignoring internal utilities [37], high average-utility itemsets [38,39], high-utility occupancy itemsets [40,41], significant utility discriminative itemsets [42], and solving HUIM in incremental [43,44] or time-stamped data [45], and so on.

However, none of the abovementioned studies considered the investment associated with itemsets during the mining process. As a result, while they identified itemsets with high utilities, they failed to reflect the true efficiency of these itemsets. This oversight limits decision-making for maximizing profit, as it disregards the investment values of the itemsets.

To address this limitation, the problem of high-efficiency itemset mining (HEIM) [23] was recently introduced, considering both utility and investment. According to HEIM, the importance of an itemset is determined by dividing its utility by its investment, defining this ratio as efficiency. HEIM aims to find all itemsets whose efficiency meets a user-defined threshold. However, due to the nature of the efficiency calculation, the efficiency measure does not exhibit monotonic or anti-monotonic properties. In other words, the efficiency values of the itemsets do not allow predicting in advance whether their supersets (or subsets) will be efficient, which complicates the problem computationally, due to the large search space. The first solution to this problem, the HEPM [23] algorithm, introduced an upper-bound called an efficiency upper-bound (*eub*), which overestimates the efficiency values of itemsets to satisfy anti-monotonicity, aiding in pruning the search space. HEPM uses a level-based candidate generation and testing strategy, conducting multiple database scans to generate candidates and filter them by calculating their actual efficiency, making it time-consuming due to the extensive number of scans and candidates. The working principle of HEPM is as follows. It employs a two-phased approach to identify *HEIs* in a dataset. In Phase 1, the algorithm explores the search space using a breadth-first search strategy. It begins by scanning the dataset to collect the *eub* values for each item. Items whose *eub* values meet a given *minE* threshold are identified as candidate 1-itemsets. Subsequently, the algorithm iteratively generates candidate 2-itemsets from 1-itemsets, candidate 3-itemsets from 2-itemsets, and so on, continuing until no new candidates can be generated. In each iteration, the algorithm produces candidate $(k + 1)$-itemsets based on the current k-itemsets. A new $(k + 1)$-itemset is formed by appending the last item of

one k-itemset to another k-itemset, provided they share the same $(k-1)$ items. During the examination of the search space, the newly generated candidates are filtered based on their *eub* values. The final result of Phase 1 is the complete set of candidate itemsets. In Phase 2, the algorithm scans the database for each candidate itemset to calculate its actual efficiency. Itemsets having efficiencies that meet the $minE$ threshold are returned as $HEIs$, which constitute the final output.

To overcome the issues seen in HEPM, the HEPMiner [23] algorithm was developed, utilizing a compact-list structure called an efficiency-list (EL). The EL of items stores the necessary information to mine $HEIs$. The EL of a k-itemset, where $k \geq 2$, is constructed by joining the ELs of two $(k-1)$-itemsets that share the same prefix itemset. HEPMiner further prunes the search space using additional upper-bounds, such as *seub* and *seubw*, alongside a matrix called an estimated efficiency co-occurrence structure ($EECS$) for storing *eub* values of each 2-itemset. The working principle of HEPMiner is as follows. First, it calculates the *eub* of each item by scanning the database once. Then, it compares the *eub* value of each item with a given $minE$ threshold. In the second database scan, it disregards the items that do not meet the threshold and sorts the remaining items in each transaction alphabetically. During this process, it generates an EL list for each remaining item and an $EECS$ structure to store the *eub* values of 2-itemsets. Afterward, the algorithm starts exploring the search space. For each itemset it visits in the search space, it first checks whether it is a HEI. Then, it calculates the *seub* value of the itemset and decides whether the extensions should be pruned. If the extensions are to be explored, it constructs EL lists for all 1-item extensions of the itemset. It discards the extensions with a *seubw* value lower than $minE$. It also decides whether EL lists need to be constructed for itemsets based on the corresponding value stored in the $EECS$ structure. The algorithm then continues exploring the search space recursively using a depth-first search strategy, along with the newly constructed EL lists. Using the values stored in the EL lists, the algorithm can easily calculate the efficiency of the itemsets, *seub* and *seubw*. For more details on the *seub*, *seubw*, EL, and $EECS$ structures and their construction, please refer to the original paper [23]. Although HEPMiner is more efficient than HEPM, it suffers from costly join operations required during the construction of lists. Subsequently, the MHEI [24] algorithm was introduced, employing a depth-first search with horizontal database representation. MHEI reduces database scanning costs through database projection and transaction merging, storing transaction identifiers for each item in the projected databases. It also introduced four upper-bounds, called sub-tree efficiency (sef), stricter sub-tree efficiency ($ssef$), local efficiency (lef), and stricter local efficiency ($slef$), to enhance pruning effectiveness. The MHEI algorithm operates as follows. First, it performs a database scan to calculate the *eub* values for each item and eliminates items whose *eub* values do not meet the $minE$ threshold, as in HEPMiner. The remaining promising items are then sorted based on a predefined order. During the second database scan, the algorithm considers only the promising items and reorganizes transactions according to the specified order of the items. If any transaction becomes empty after removing unpromising items, it is removed from the database. The rearranged transactions are then subjected to a sorting process to keep similar transactions close to each other. This facilitates the merging of identical projected transactions in later steps. The algorithm then scans the reorganized database, calculating the $ssef$ values for each item and storing the IDs of the transactions it finds in a list called CID. Accordingly, items whose $ssef$ values do not satisfy the $minE$ threshold are eliminated. Following this, MHEI performs a depth-first search to explore the search space. This is an iterative process where a prefix itemset, X (initially an empty set), is extended by adding a single item and evaluating the potential for further extensions. For each single-item extension of X with an item i, denoted as $Z = \{X \cup i\}$, the projected database of X is scanned to

compute the efficiency value of Z and obtain its projected dataset. If the efficiency value of Z satisfies the $minE$ threshold, Z is identified as a HEI. Subsequently, for the per item v that follows i in the processing order, the values $ssef(X, v)$ and $slef(X, v)$, as well as the CID of v within the projected database of Z, are obtained. Any item (along with Z) whose $ssef$ value satisfies $minE$ is considered a potential single-item extension of Z, while items with $slef$ values satisfying $minE$ are considered for future extensions of Z. Extensions with $ssef$ and $slef$ values below $minE$ are discarded. The new prefix becomes Z, and the same process is repeated until the exploration procedure terminates. Note that, to reduce database scanning costs, the algorithm applies a transaction merging step to each projected database. Additionally, the CID of itemsets is only used to examine the required transactions, which provides further time savings. For detailed information on the calculation of upper-bounds, transaction merging, and database projection, please refer to the original paper [24]. MHEI outperforms both HEPM and HEPMiner in runtime, memory consumption, and the number of generated candidates [24]. Additionally, the HEIM problem has been extended to the high-average-efficiency itemset mining (HAEIM) problem, which more fairly evaluates the efficiency of itemsets of different lengths [46]. In HAEIM, the importance of itemsets is determined using a metric called average-efficiency. The average-efficiency of an itemset is calculated by dividing its efficiency by its length (the number of items in the itemset).

However, all existing HEIM algorithms assume that databases contain only positive utilities, leading to incomplete discovery of $HEIs$ when applied to databases that also contain negative utilities. The primary reason for this is that the upper-bound models used by existing algorithms may underestimate the actual efficiency values of itemsets in the presence of negative utilities. This causes the search space to be pruned incorrectly, missing important $HEIs$, and provides decision-makers with incomplete insights, which can hinder their ability to make informed decisions. To address this limitation, this study focuses on developing techniques for the complete and correct discovery of $HEIs$ in databases that contain both positive and negative utilities. This ensures that decision-makers are provided with comprehensive and reliable data, allowing them to optimize future strategies based on a full understanding of past behaviors. This is especially valuable in real-world applications, where negative utility can represent costs or losses, as seen in retail promotions, bundled services, or investment strategies.

3. Preliminaries and Problem Statement

This section covers the basic concepts of HEIM problem. It also discusses the limitations the existing HEIM algorithms face when they are applied to databases with negative utility values.

Consider a set of items $I = \{i_1, i_2, \ldots, i_n\}$, where each item has an external utility $eu(i)$ and a unit investment value $uInv(i)$. A transactional database DB is defined as a collection of transactions: $DB = \{T_1, T_2, \ldots, T_m\}$. Each transaction T_j in DB can contain any subset of the items in I, and for each item i in T_j, there is an internal utility $iu(i, T_j)$.

For example, consider the sample database DB shown in Table 1. It consists of eight transactions and contains the set of items $I = \{a, b, c, d, e, f, g\}$. For example, in the first transaction, T_1, the items a, c, d, and f appear with internal utilities of 5, 4, 1, and 2, respectively. Thus, $iu(a, T_1) = 5$ $iu(c, T_1) = 4$, $iu(d, T_1) = 1$, and $iu(f, T_1) = 2$. The external utility and unit investment values of the items are provided in Table 2. For example, the external utility of item a is $eu(a) = 1$, and its unit investment is $uInv(a) = 2$.

The external utility of an item can be either positive or negative. As shown in Table 2, items a, b, d, and g have positive external utilities, while the remaining items have negative external utilities. Note that the terms "positive items" and "negative items" will be used

to denote items with positive and negative external utilities, respectively, throughout this paper.

Table 1. A sample transactional database.

TID	Items	Internal Utilities
T_1	a, c, d, f	5, 4, 1, 2
T_2	b, e	2, 2
T_3	a, c, d, e, f, g	3, 6, 4, 1, 1, 1
T_4	c, g	1, 7
T_5	a, d, e, f	2, 3, 1, 2
T_6	b, c, e, g	1, 2, 2, 1
T_7	e, f	1, 1
T_8	a, d, e, f, g	2, 2, 1, 1, 1

Table 2. External utilities (profits) for items.

Item	a	b	c	d	e	f	g
External utility	1	2	−3	5	−2	−1	2
Unit investment	2	3	4	5	6	4	13

Definition 1 (**Total investment [23]**). *The total investment of an item i in a given database DB, denoted as $inv(i)$, is defined as*

$$inv(i) = uInv(i) \times \sum_{i \in T_j \wedge T_j \in DB} iu(i, T_j). \tag{1}$$

The total investment of an itemset X in a given database DB, denoted as $inv(i)$, is defined as

$$inv(X) = \sum_{i_k \in X} inv(i_k). \tag{2}$$

For example, item a appears in transactions T_1, T_3, T_5, and T_8. Therefore, the total investment of item a, $inv(a)$, is calculated as $inv(a) = uInv(a) \times (iu(a, T_1) + iu(a, T_3) + iu(a, T_5) + iu(a, T_8)) = 2 \times (5 + 3 + 2 + 2) = 24$. The total investment value of each item is given in Table 3. As another example, the total investment of the itemset $\{a, d, f\}$ is calculated as $inv(\{a, d, f\}) = inv(a) + inv(d) + inv(f) = 24 + 50 + 28 = 102$.

Table 3. Total investment values for items.

Item	a	b	c	d	e	f	g
Total investment	24	9	52	50	48	28	130

Definition 2 (**Utility [14]**). *The utility of an itemset X in a given transaction T_j, where $X \subseteq T_j$, denoted as $u(X, T_j)$, is defined as*

$$u(X, T_j) = \sum_{i_k \in X} iu(i_k, T_j) \times eu(i_k). \tag{3}$$

The utility of an itemset X in a given database DB, denoted as $u(X)$, is defined as

$$u(X) = \sum_{X \subseteq T_j \wedge T_j \in DB} u(X, T_j). \tag{4}$$

For example, consider the itemset $\{a, d, f\}$, which appears in transactions T_1, T_3, T_5, and T_8. The utility of $\{a, d, f\}$ in T_1 is calculated as $u(\{a, d, f\}, T_1) = iu(a, T_1) \times eu(a) + iu(d, T_1) \times eu(d) + iu(f, T_1) \times eu(f) = 5 \times 1 + 1 \times 5 + 2 \times (-1) = 8$. Similarly, the utilities of $\{a, d, f\}$ in transactions T_3, T_5, and T_8 are obtained as 22, 15, and 8, respectively. Consequently, the utility of $\{a, d, f\}$ in the database DB is $u(\{a, d, f\}) = 8 + 22 + 15 + 11 = 56$.

Definition 3 (**Efficiency of an itemset [23]**). *The efficiency of an itemset X in a given database DB, denoted as $e(X)$, is defined as*

$$e(X) = u(X)/inv(X). \tag{5}$$

For example, the efficiency of the itemset $\{a, d, f\}$ is calculated as $e(\{a, d, f\}) = 56/102 = 0.5490$.

Definition 4 (**High-efficiency itemset [23]**). *An itemset X is classified as a high-efficiency itemset (HEI) if its efficiency is not lower than a specified minimum efficiency threshold ($minE$).*

For example, if $minE$ is set to 0.35, then the itemset $\{a, d, f\}$ is a HEI, since $e(\{a, d, f\}) = 0.5490 \geq minE = 0.35$. Table 4 summarizes all the itemsets that are $HEIs$ within the sample DB presented in Table 1, when $minE = 0.35$.

Table 4. High-efficiency itemsets when $minE$ is set to 0.35.

Itemset	Efficiency
$\{a\}$	0.5
$\{b\}$	0.6667
$\{d\}$	1
$\{a, d\}$	0.8378
$\{d, e\}$	0.3980
$\{d, f\}$	0.5641
$\{a, d, e\}$	0.3770
$\{a, d, f\}$	0.5490

However, solving the HEIM problem is challenging and complex, due to its large search space. Furthermore, the efficiency of itemsets cannot be used to reduce the search space, since the efficiency measure does not have an anti-monotonic (or monotonic) property. To address this, researchers [23,24] have focused on developing upper-bounds that overestimate the efficiency values of itemsets, providing the anti-monotonic property needed to effectively prune the search space. On the other hand, the existing upper-bounds were developed under the assumption that databases contain only positive items. When databases also include negative items, these upper-bounds may underestimate the efficiency of itemsets, leading to incorrect pruning of the search space. Consequently, applying the existing HEIM algorithms to databases with negative items may result in incomplete discovery of $HEIs$. To better understand this issue, let us consider one of the existing upper-bounds, called the efficiency upper-bound (eub) [23], which is utilized by all existing HEIM algorithms to prune the search space. The details of the eub model are as follows.

Definition 5 (**Efficiency upper-bound (eub) [23]**). *Let the transaction utility (tu) of a transaction T_j be defined as the sum of the utility values of all items within that transaction, given by*

$$tu(T_j) = \sum_{i_k \in T_j} u(i_k, T_j). \tag{6}$$

Let the transaction weighted utility (twu) of an itemset X be the sum of the tu values of all transactions that contain X, defined as

$$twu(X) = \sum_{X \subseteq T_j \wedge T_j \in DB} tu(T_j). \tag{7}$$

The efficiency upper-bound of X is defined as the ratio of the twu to the total investment of X, and it is given by

$$eub(X) = twu(X)/inv(X). \tag{8}$$

Based on the anti-monotonic property of *eub*, for any itemset X and its supersets X', the inequality $e(X') \leq eub(X') \leq eub(X)$ holds [23]. This implies that $eub(X)$ serves as an overestimation of the efficiency of X and its supersets. Accordingly, X or any of its supersets cannot be a HEI if $eub(X) < minE$, and, therefore, they can be pruned from the search space without further examination.

However, in databases with negative items, the anti-monotonic property of *eub* may no longer hold. In other words, for any itemset X, $eub(X)$ might underestimate the efficiency of X or any of its supersets. For example, let us consider the itemset $\{a, d, f\}$. This itemset appears in transactions T_1, T_3, T_5, and T_8. The transaction utility of T_1 is calculated as $tu(T_1) = u(a, T_1) + u(c, T_1) + u(d, T_1) + u(f, T_1) = 5 \times 1 + 4 \times (-3) + 1 \times 5 + 2 \times (-1) = -4$. For the other transactions, $tu(T_3) = 4$, $tu(T_5) = 13$, and $tu(T_8) = 11$. Thus, $twu(\{a, d, f\}) = tu(T_1) + tu(T_3) + tu(T_5) = 14 + tu(T_8) = -4 + 4 + 13 + 11 = 24$. Consequently, $eub(\{a, d, f\}) = twu(\{a, d, f\})/inv(\{a, d, f\}) = 24 / 102 = 0.2353$. Accordingly, itemset $\{a, d, f\}$ or any of its supersets cannot be classified as a HEI when $MinE = 0.35$ and can thus be pruned based on $eub(\{a, d, f\}) = 0.2353$. However, $e(\{a, d, f\}) = 0.5490 \geq 0.35$, so itemset $\{a, d, f\}$ is a HEI. As another example, consider the item b. $eub(b) = (twu(b) = (tu(T_2) + tu(T_6))/inv(b) = (0 + (-6))/9 = -0.6667$. In this case, no matter how low $minE$ is, item b or any of its supersets can never be classified as a HEI according to the $eub(b)$. However, item b is also a HEI when $minE = 0.35$.

4. Mining High-Efficiency Itemsets with Negative Utilities

This study introduces an algorithm named MHEINU (mining high-efficiency itemsets with negative utilities), specifically designed to extract $HEIs$ from databases also containing negative utilities. The algorithm employs two upper-bounds, each integrated with an effective pruning strategy, and utilizes a list-based data structure to enhance performance.

The first part of this section introduces two new upper-bounds to safely reduce the search space of the HEIM problem with negative utilities. The next subsection describes a list-based data structure called $ELNU$ (efficiency-list with negative utilities), which stores the information necessary for extracting $HEIs$. The subsequent subsection discusses the set-enumeration tree of the search space of the problem. The following subsection outlines the overall process of the proposed MHEINU algorithm, including its pseudo-code. The next subsection addresses the correctness and completeness of the proposed MHEINU algorithm. Finally, the last subsection provides an execution trace of the algorithm using an illustrative example.

4.1. Proposed Upper-Bounds

This study introduces two upper-bounds, along with corresponding pruning strategies, to efficiently and safely reduce the search space, thereby enhancing the mining process of the HEIM problem with negative utilities.

The first upper-bound, called the upper-bound efficiency with negative utilities (*uben*), is designed to determine whether an itemset and its supersets can contain any *HEIs*. Details are provided below.

Definition 6 (**Positive utility of a transaction**). *The positive utility of a transaction T_j, denoted as $pu(T_j)$, is defined as*

$$pu(T_j) = \sum_{i_k \in T_j \wedge eu(i_k) \geq 0} u(i_k, T_j). \tag{9}$$

For example, consider transaction T_1. It contains two positive items, a and d. The positive utility of T_1 is calculated as $pu(T_1) = u(a, T_1) + u(d, T_1) = 10$. Table 5 presents the positive utility of each transaction.

Table 5. Positive utility value of each transaction.

Transaction	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Positive Utility	10	4	25	14	17	4	0	14

Note that, since the values of *pu* obtained from the transactions provide an overestimate of the utilities of the itemsets (i.e., $u(X, T_j) \leq pu(X, T_j)$ is clear for any itemset X, where $X \subseteq T_j$, these values can be directly used in an upper-bound designed for the efficiency values of the itemsets. However, the existence of negative utilities allows for a tighter upper-bound design, as presented in the following definitions.

Definition 7 (**Positive utility upper-bound**). *The positive utility upper-bound (pub) of an item i in a transaction T_j, where $i \in T_j$, denoted as $pub(i, T_j)$, is defined as*

$$pub(i, T_j) = \begin{cases} pu(T_j), & \text{if } u(i, T_j) \geq 0, \\ pu(T_j) + u(i, T_j), & \text{else if } pu(T_j) + u(i, T_j) > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

The pub of an itemset X in a transaction T_j, where $X \subseteq T_j$, denoted as $pub(X, T_j)$, is defined as

$$pub(X, T_j) = \min_{i_k \in X} pub(i_k, T_j). \tag{11}$$

The pub of an itemset X in a database DB, denoted as $pub(X)$, is defined as

$$pub(X) = \sum_{X \subseteq T_j \wedge T_j \in DB} pub(X, T_j). \tag{12}$$

For example, item a appears in transactions T_1, T_3, T_5, and T_8. Since a is a positive item, the *pub* values are calculated as follows: $pub(a, T_1) = pu(T_1) = 10$, $pub(a, T_3) = pu(T_3) = 25$, $pub(a, T_5) = pu(T_5) = 17$, and $pub(a, T_8) = pu(T_8) = 14$. As a result, $pub(a) = 10 + 25 + 17 + 14 = 66$. As another example, item c appears in transactions T_1, T_3, T_4, and T_6. Since c is a negative item, it is necessary to check whether $pu(T_j) + u(c, T_j) > 0$ for each transaction T_j containing c to determine $pub(c, T_j)$. For transactions T_1 and T_6, $pu(T_1) + u(c, T_1) = 10 + (-12) \leq 0$ and $pu(T_6) + u(c, T_6) = 4 + (-6) \leq 0$, so $pub(c, T_1) = pub(c, T_6) = 0$. For transactions T_3 and T_4, $pu(T_3) + u(c, T_3) = 25 + (-18) = 7 > 0$ and $pu(T_4) + u(c, T_4) = 14 + (-3) = 11 > 0$, so $pub(c, T_3) = 7$ and $pub(c, T_4) = 11$. Consequently, $pub(c) = 0 + 7 + 11 + 0 = 18$. The *pub* values for each item in each transaction are summarized in Table 6. Now, let us calculate *pub* for the itemset $\{a, d, f\}$. It appears in transactions T_1, T_3, T_5, and T_8. $pub(\{a, d, f\}, T_1)$ is equal to $min(pub(a, T_1), pub(d, T_1), pub(f, T_1)) = min(10, 10, 8) = 8$.

In a similar way, $pub(\{a,d,f\}, T_3) = 24$, $pub(\{a,d,f\}, T_5) = 15$, and $pub(\{a,d,f\}, T_8) = 13$. Therefore, $pub(\{a,d,f\}) = 8 + 24 + 15 + 13 = 60$. For another itemset, $\{a,c\}$, the *pub* value is calculated as $pub(\{a,c\}) = pub(\{a,c\}, T_1) + pub(\{a,c\}, T_3) = min(10, 0) + min(25, 7) = 7$.

Table 6. The *pub* values of items in each transaction.

Item	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
a	10	–	25	–	17	–	–	14
b	–	4	–	–	–	4	–	–
c	0	–	7	11	–	0	–	–
d	10	–	25	–	17	–	–	14
e	–	0	23	–	15	0	0	12
f	8	–	24	–	15	–	0	13
g	–	–	25	14	–	4	–	14

Theorem 1 (For any itemset, its *pub* is not lower than its utility)**.** *For any itemset X, the inequality $u(X) \leq pub(X)$ holds.*

Proof of Theorem 1. For any itemset X in a transaction T_j, such that $X \subseteq T_j$, the following cases apply:

- Case 1: X contains only positive items. In this case, $pub(X, T_j) = pu(T_j)$. Therefore, $u(X, T_j) \leq pub(X, T_j)$ holds because it is clear that $u(X, T_j)$ can be at most equal to $pu(T_j)$.
- Case 2: X contains both positive and negative items. Let $X.NI = \{n_1, n_2, \cdots, n_m\}$ denote the set of negative items within X. In this case, $u(X, T_j) \leq pub(X, T_j)$ holds based on the following:
  - If $pu(T_j) + u(n_k, T_j) > 0$ is correct for each $n_k \in X.NI$, then $pub(X, T_j) = \min_{n_k \in T_j}(pu(T_j) + u(n_k, T_j))$. Thus, $u(X, T_j) \leq pub(X, T_j)$ holds because it is clear that $u(X, T_j)$ can be at most equal to $\min_{n_k \in T_j}(pu(T_j) + u(n_k, T_j))$.
  - Otherwise, i.e., $pu(T_j) + u(n_k, T_j) \leq 0$ is correct for any $n_k \in X.NI$, $pub(X, T_j) = 0$. Therefore, $u(X, T_j) < pub(X, T_j)$ holds because it is clear that $u(X, T_j) \leq 0$.
- Case 3: X contains only negative items. In this case, $pub(X, T_j) = 0$. Therefore, $u(X, T_j) < pub(X, T_j)$ holds because $u(X, T_j) < 0$ is obvious.

Consequently, for any transaction T_j containing X, $u(X, T_j) \leq pub(X, T_j)$ holds. Thus, it is concluded that $u(X) \leq pub(X)$. □

Theorem 2 (Anti-monotonic property of pub)**.** *For any itemset X and its superset XS, the pub of XS is always less than or equal to the pub of X. Therefore, the inequality $pub(XS) \leq pub(X)$ holds.*

Proof of Theorem 2. Let $T(X)$ and $T(XS)$ denote the sets of transactions containing X and XS, respectively, in a given database DB. Since $X \subseteq XS$, $pub(XS, T_j) = \min_{i_k \in XS} pub(i_k, T_j)$ $\leq pub(X, T_j) = \min_{i_k \in X} pub(i_k, T_j)$ is correct for any transaction $T_j \in T(XS)$. Additionally, it is clear that $T(XS) \subseteq T(X)$ is clear. Thus, $pub(XS) \leq pub(X)$ holds. □

Based on the above definitions and properties, *uben* is proposed as follows.

Definition 8 (**Upper-bound efficiency with negative utilities, uben**)**.** *The uben of an itemset X in a given database DB, denoted as $uben(X)$, is defined as*

$$uben(X) = pub(X)/inv(X). \tag{13}$$

For example, the *uben* of the itemset $\{a, d, f\}$ is calculated as $uben(\{a, d, f\}) = pub(\{a, d, f\})/inv(\{a, d, f\}) = 60 \ / \ 102 = 0.5882$.

Theorem 3 (For any itemset, its *uben* is not lower than its efficiency)**.** *For any itemset X, the inequality $e(X) \leq uben(X)$ holds.*

Proof of Theorem 3. $e(X) = u(X) \ / \ inv(X) \leq uben(X) = pub(X) \ / \ inv(X)$ holds, since we have $u(X) \leq pub(X)$ based on Theorem 1. □

Theorem 4 (Anti-monotonic property of uben)**.** *For any itemset X and its superset XS, the uben of XS is always less than or equal to the uben of X. Therefore, the inequality $uben(XS) \leq uben(X)$ holds.*

Proof of Theorem 4. $uben(XS) = pub(XS) \ / \ inv(XS) \leq uben(X) = pub(X) \ / \ inv(X)$ holds for any itemset X and its superset XS, because we have $pub(XS) \leq pub(X)$ based on Theorem 2 and $inv(XS) \geq inv(X)$ is clear. □

Pruning Strategy 1 (Pruning with uben). *Based on Theorems 3 and 4 and their proofs, if $uben(X) < minE$, then X or any of its supersets cannot be a HEI. Therefore, they can safely be pruned from the search space.*

For example, the *uben* of the itemset $\{a, c\}$ is calculated as follows. The transactions in which both a and c appear together are T_1 and T_3. As shown in Table 6, the values are $pub(a, T_1) = 10$, $pub(c, T_1) = 0$, $pub(a, T_3) = 25$, and $pub(c, T_3) = 7$. Additionally, $inv(a) = 24$ and $inv(c) = 52$. Thus, $uben(\{a, c\}) = pub(\{a, c\})/inv(\{a, c\}) = (pub(\{a, c\}, T_1) + pub(\{a, c\}, T_3))/inv(\{a, c\}) = (min(10, 0) + min(25, 7))/(24 + 52) = 0.0921$. If the minimum efficiency threshold is set to any value greater than 0.0921, then based on the *uben* value of $\{a, c\}$, it is guaranteed that neither $\{a, c\}$ nor any of its supersets will be a HEI. Therefore, the mining process can be carried out without the need to calculate the efficiency values for these itemsets.

However, if an itemset X shows potential for producing a HEI based on $uben(X)$ (i.e., if $uben(X) \geq minE$), then the search space cannot be pruned using $uben(X)$. On the other hand, some of the supersets of X may not qualify as $HEIs$. Therefore, it is crucial to develop additional upper-bounds to further prune the search space and enhance the efficiency of solving the problem. To address this issue, this paper introduces an additional upper-bound, the details of which are given below.

Definition 9 (Upper-bound efficiency with negative utilities using an item, ubeni). *The upper-bound efficiency with negative utilities of an itemset X using an item y, where $y \notin X$ denoted as $ubeni(X, y)$, is defined as*

$$ubeni(X, y) = min(pub(X), pub(y))/(inv(X) + inv(y)). \tag{14}$$

For example, $ubeni(\{a, d, f\}, c) = min(pub(\{a, d, f\}), pub(c)) \ / \ (inv(\{a, d, f\}) + inv(c)) = min(60, 18) \ / \ (102 + 52) = 0.1169$.

Theorem 5. *For any itemset X and an item y such that $y \notin X$, the inequality $e(\{X \cup y\}) \leq ubeni(X, y)$ holds.*

Proof of Theorem 5. $e(\{X \cup y\}) = u(\{X \cup y\})/ \ (inv(X) + inv(y)) \leq ubeni(X, y) = min(pub(X), pub(y))/ \ (inv(X) + inv(y))$ holds, since we have $u(\{X \cup y\}) \leq pub(\{X \cup y\})$

based on Theorem 1, and $pub(\{X \cup y\}) \leq pub(X)$ and $pub(\{X \cup y\}) \leq pub(y)$ based on Theorem 2. □

Theorem 6 (Anti-monotonic property of ubeni)**.** *For any itemset $Xy = \{X \cup y\}$ and its superset $Xyz = \{X \cup y \cup z\}$, where $y \notin X$ and $z \notin Xy$, the ubeni of $\{Xy, z\}$ is always less than or equal to the ubeni of $\{X, y\}$. Therefore, the following inequality holds: $ubeni(Xy, z) \leq ubeni(X, y)$.*

Proof of Theorem 6. $ubeni(Xy, z) = min(pub(Xy), pub(z))/(inv(Xy) + inv(z)) \leq ubeni(X, y) = min(pub(X), pub(y))/(inv(X) + inv(y))$ holds for any itemset Xy and its superset Xyz, because we have $pub(Xy) \leq pub(X)$ and $pub(Xy) \leq pub(y)$ based on Theorem 2, and $inv(Xy) + inv(z) > inv(X) + inv(y)$ is clear. □

Pruning Strategy 2 (Pruning with ubeni). *Based on Theorems 5 and 6 and their proofs, if $ubeni(X, y) < minE$, then $\{X \cup y\}$ or any of its extensions cannot be a HEI. Therefore, they can safely be pruned from the search space.*

For example, the *ubeni* of the itemset $\{a, e\}$ is calculated as follows. As shown in Table 6, the pub values of a in each transaction that it appears are $pub(a, T_1) = 10$, $pub(a, T_3) = 25$, $pub(a, T_5) = 17$, and $pub(a, T_8) = 14$. Meanwhile, the pub values of e in each transaction that it appears are $pub(e, T_2) = 0$, $pub(e, T_3) = 23$, $pub(e, T_5) = 15$, $pub(e, T_6) = 0$, $pub(e, T_7) = 0$, and $pub(e, T_8) = 12$. Additionally, $inv(a) = 24$ and $inv(e) = 48$. Therefore, $ubeni(\{a, e\}) = min(pub(a), pub(e))/inv(\{a, e\}) = min((10 + 25 + 17 + 14), (0 + 23 + 15 + 0 + 0 + 12))/(24 + 48) = 50/72 = 0.6944$. If the minimum efficiency threshold is set to any value greater than 0.6944, then, based on the *ubeni* value of $\{a, e\}$, it is guaranteed that none of the supersets of a including e will be a HEI. Therefore, the mining process can be carried out without the need to calculate the efficiency values for these itemsets.

4.2. The Data Structure

To effectively solve the problem of HEIM with negative utilities, it is also important to develop data structures. These structures should enable efficient computation of upper-bound and efficiency values for itemsets. Therefore, this study introduces a list-based data structure called the efficiency-list with negative utilities ($ELNU$).

Definition 10 (Efficiency-list with negative utilities)**.** *The efficiency-list with negative utilities of an itemset X, denoted as $ELNU(X)$, stores the necessary information for determining whether X is a HEI or can be pruned. $ELNU(X)$ includes an entry E of the form $\langle E.tid, E.u, E.pub \rangle$ for each transaction containing X, where $E.tid$ represents the transaction identifier, $E.u$ denotes the utility of X within that transaction, and $E.pub$ indicates the positive utility upper-bound of X in the same transaction. Additionally, $ELNU(X)$ stores the utility, positive utility upper-bound, and investment values of X as $ELNU(X).u$, $ELNU(X).pub$, and $ELNU(X).inv$, respectively.*

The $ELNUs$ of items can be easily constructed through a database scan. For example, $ELNU(b)$ can be constructed as follows. There are two transactions that include item b. The first transaction is T_2, where $u(b, T_2) = 4$ and $pu(T_2) = 4$. As a result, a new entry, such as $(2, 4, 4)$, is added to the entries of $ELNU(b)$ during the processing of T_2. Additionally, $ELNU(b).u$, $ELNU(b).pub$, and $ELNU(b).inv$ are initialized as $u(b, T_2) = 4$, $pu(T_2) = 4$, and $inv(b) = 9$, respectively. The second transaction that includes item b is T_6. Since $u(b, T_6) = 2$ and $pu(b, T_6) = 4$, an another entry, such as $(6, 2, 4)$, is added to the entries of $ELNU(b)$ during the processing of T_6. Furthermore, $ELNU(b).u$ and $ELNU(b).pub$ are updated to $4 + 2 = 6$ and $4 + 4 = 8$, respectively. Figure 1 illustrates the $ELNU$ of each item for the running example.

The $ELNUs$ of longer itemsets can be easily constructed by joining the $ELNUs$ of shorter itemsets. To construct the $ELNU$ of an itemset $Xy = \{X \cup y\}$ where $y \notin X$, using the $ELNU$ of itemset X and the $ELNU$ of item y, the process begins by initializing $ELNU(Xy)$ with the following values: $ELNU(Xy).inv = ELNU(X).inv + ELNU(y).inv$, $ELNU(Xy).u = 0$, $ELNU(Xy).pub = 0$, and $ELNU(Xy).Entries = \emptyset$. Next, for each pair of entries $E_X \in ELNU(X)$ and $E_y \in ELNU(y)$ that share the same tid, a new entry E is generated with $E.tid = E_X.tid$, $E.u = E_X.u + E_y.u$, and $E.pub = min(E_X.pub, E_y.pub)$, which is then added to $ELNU(Xy).Entries$. Throughout this process, the values of $ELNU(Xy).u$ and $ELNU(Xy).pub$ are updated based on the corresponding $E.u$ and $E.pub$ values. For example, the construction process of $ELNU(\{a,c\})$ using $ELNU(a)$ and $ELNU(c)$ is illustrated in Figure 2.

Tid	*u*	*pub*
1	5	10
3	3	25
5	2	17
8	2	14
u : 12		
pub : 66		
inv : 24		

(**a**) $ELNU(a)$

Tid	*u*	*pub*
2	4	4
6	2	4
u : 6		
pub : 8		
inv : 9		

(**b**) $ELNU(b)$

Tid	*u*	*pub*
1	−12	0
3	−18	7
4	−3	11
6	−6	0
u : −39		
pub : 18		
inv : 52		

(**c**) $ELNU(c)$

Tid	*u*	*pub*
1	5	10
3	20	25
5	15	17
8	10	14
u : 50		
pub : 66		
inv : 50		

(**d**) $ELNU(d)$

Tid	*u*	*pub*
2	−4	0
3	−2	23
5	−2	15
6	−4	0
7	−2	0
8	−2	12
u : −16		
pub : 50		
inv : 48		

(**e**) $ELNU(e)$

Tid	*u*	*pub*
1	−2	8
3	−1	24
5	−2	15
7	−1	0
8	−1	13
u : −7		
pub : 60		
inv : 28		

(**f**) $ELNU(f)$

Tid	*u*	*pub*
3	2	25
4	14	14
6	2	4
8	2	14
u : 20		
pub : 57		
inv : 130		

(**g**) $ELNU(g)$

Figure 1. $ELNU$ of each item for the running example.

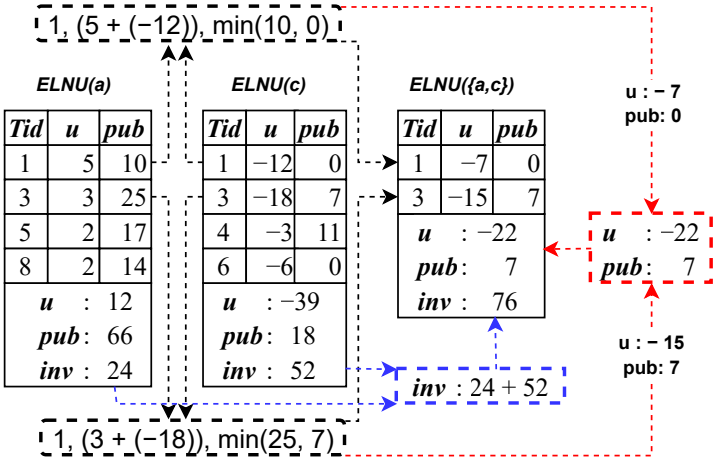


Figure 2. The construction process of $ELNU(\{a,c\})$.

4.3. The Set-Enumeration Tree of the Search Space

The search space for the HEIM problem can be represented as a set-enumeration tree, arranged according to any given order of the items. Each node in this tree corresponds to a distinct itemset derived from the items. In theory, for a set of I items, the tree contains $2^I - 1$ nodes. Previous itemset mining studies [14,24,27,39] have shown that choosing an appropriate processing order for items can significantly reduce the size of the enumeration tree.

In this study, the total processing order ($\prec$) of the items is determined as the *uben*-ascending order. For example, Table 7 provides the *uben* value for each item in the running example. Therefore, the total processing order of the items is obtained as $c \prec g \prec b \prec e \prec d \prec f \prec a$ since $uben(c) < uben(g) < uben(b) < uben(e) < uben(d) < uben(f) < uben(a)$.

Accordingly, Figure 3 depicts the set-enumeration tree of the search space for the running example. Due to space limitations, the tree depicted in the figure is not fully expanded.

Table 7. *uben* values of items.

Item	*a*	*b*	*c*	*d*	*e*	*f*	*g*
uben	2.75	0.8889	0.3462	1.32	1.0417	2.1429	0.4385

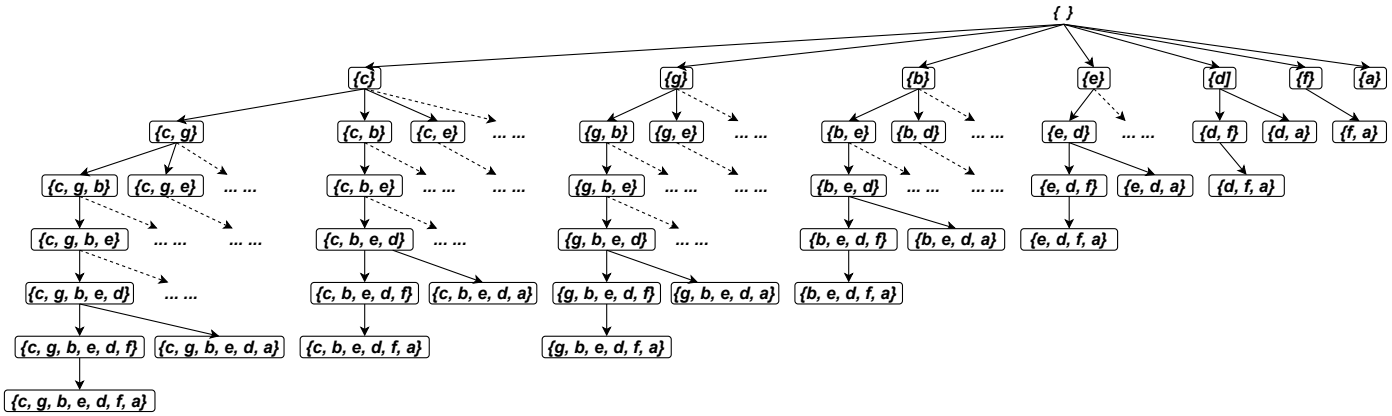


Figure 3. The set-enumeration tree of the search space for the running example.

4.4. Algorithmic Description of the Proposed MHEINU Algorithm

This section outlines the overall mining process of the proposed MHEINU algorithm. The pseudo-code for the MHEINU algorithm is presented in Algorithm 1. The input parameters include a database DB containing transactions, along with the utility and investment values of items, and a user defined minimum efficiency threshold $minE$. The output is the complete and correct set of $HEIs$ based on the given $minE$. The MHEINU algorithm performs the mining task as follows. It begins by obtaining the $ELNU$ for each item i within the database, collecting them into a set called $ELNUS$ (Line 1). Next, it sorts the $ELNUs$ of items in ascending order of their *uben* values (Line 2). It then iterates over the $ELNU$ of each item in the sorted $ELNUS$ (Line 3). For each item i, it checks if the *uben* value meets or exceeds the given $minE$, to determine whether item i can be pruned based on the Pruning Strategy 1 (Line 4). If condition $uben(i) \geq minE$ is satisfied, the algorithm checks if the item i is a HEI and then proceeds to explore its extensions in the search space by invoking the Search algorithm (Lines 5 to 7).

Algorithm 1: MHEINU

Input : DB, a database; $minE$, a minimum efficiency threshold.
Output: $HEIs$, the complete set of discovered high-efficiency itemsets.

```
1 Obtain the ELNU(i) of each item i ∈ DB and put them into a set ELNUS;
2 Sort ELNUs of items in ELNUS based on uben-ascending order;
3 foreach item ELNU(i) ∈ ELNUS do
4     if uben(i) ≥ minE then
5         if e(i) ≥ minE then
6             Output i as a HEI;
7         Search (ELNU(i), ELNUS, minE);
```

Algorithm 2 provides the pseudo-code for the Search algorithm, which iteratively examines the sub-tree of a given itemset X in the search space. The inputs of the algorithm include $ELNU(X)$, $ELNUS$, and $minE$. Its purpose is to identify and output the set of $HEIs$ that exist in the sub-tree of X. The algorithm works as follows. For each item y, where each

item $x \in X$ precedes y in the order $\prec$, the algorithm checks whether $ubeni(X, y)$ is greater than or equal to the given $minE$ (Line 2). If this condition is not met, the algorithm applies the Pruning Strategy 2 and continues with the next item that follows y according to the $\prec$ order. Otherwise, the algorithm constructs the $ELNU$ for the new itemset $Xy = \{X \cup y\}$ by invoking the Construction Algorithm (Line 3). Once the $ELNU(Xy)$ has been constructed, it evaluates whether $uben(Xy)$ meets the given $minE$ (Line 4). If not, it applies the Pruning Strategy 1. If the $uben(Xy) \geq minE$ condition is met, the algorithm checks if Xy qualifies as a HEI and, if so, outputs it as a HEI (Lines 5–6). Following this, it calls itself to explore the sub-tree of the current itemset Xy (Line 7). This iterative and recursive process is repeated until no further $ELNUs$ that can be generated, thus completing the process of discovering extensions of the given itemset X.

Algorithm 2: Search

Input : $ELNU(X)$, $ELNU$ of an itemset X; $ELNUS$, the set of $ELNUs$ of items; $minE$, a minimum efficiency threshold percent

Output: The complete set of $HEIs$ which are extensions of X.

```
1 foreach ELNU(y) ∈ ELNUS, where ∀ x ∈ X ≺ y do
2     if ubeni(X, y) ≥ minE then
3         ELNU(Xy) ← Construct (ELNU(X), ELNU(y));
4         if uben(Xy) ≥ minE then
5             if e(Xy) ≥ minE then
6                 Output Xy as a HEI;
7             Search (ELNU(Xy), ELNUS, minE);
```

Algorithm 3 provides the pseudo-code for the Construct Algorithm. It receives two inputs: $ELNU(X)$, representing the $ELNU$ of the itemset X to be extended, and $ELNU(y)$, representing the $ELNU$ of the item y that can extend X. The goal of the algorithm is to generate and return the $ELNU$ of the itemset $Xy = \{X \cup y\}$. The algorithm begins by initializing the output $ELNU(Xy)$ as an empty list (Line 1). It then calculates the investment of Xy (Line 2). Two indices, i and j, are initialized to 0 (Line 3). These indices will serve as pointers to track the current entries in $ELNU(X)$ and $ELNU(y)$, respectively. The main processing occurs within a while loop, which continues until all entries in either $ELNU(X)$ or $ELNU(y)$ have been processed (Lines 4–14). During each iteration, the algorithm compares the current entries from $ELNU(X)$ and $ELNU(y)$, as indicated by the indices i and j, to determine if they share the same tid. If the tid values match (Line 7), a new entry is generated and added to the entries of $ELNU(Xy)$ (Lines 8–9). The utility and positive utility upper-bounds of Xy are then updated based on the values of the newly generated entry (Lines 10–11). Additionally, the indices i and j are incremented to proceed to the next entries in $ELNU(X)$ and $ELNU(y)$ (Line 12). If the current entries from $ELNU(X)$ and $ELNU(y)$ do not share the same tid value, the algorithm increments the index corresponding to the entry with the smaller tid value: if the tid of E_X is lower than the tid of E_y, index i is incremented (Line 13); otherwise, index j is incremented (Line 14). Finally, the algorithm returns the constructed $ELNU(Xy)$ (Line 15).

Algorithm 3: Construct ($ELNU(X)$, $ELNU(y)$)

Input : $ELNU(X)$, $ELNU$ of an itemset X; $ELNU(y)$, $ELNU$ of an item y.
Output: $ELNU(Xy)$, $ELNU$ of an itemset Xy.

$ELNU(Xy) = \emptyset$;
$ELNU(Xy).inv = ELNU(X).inv + ELNU(y).inv$;
$i, j = \emptyset$;
while $i < |ELNU(X).Es| \wedge j < |ELNU(y).Es|$ **do**
  $E_X = ELNU(X).Es[i]$;
  $E_y = ELNU(y).Es[j]$;
  if $Ex.tid == Ey.tid$ **then**
    $E_{new} = (E_X.tid, E_X.u + E_y.u, min(E_X.pub, E_y.pub))$;
    $ELNU(Xy).Es = ELNU(Xy).Es \cup E_{new}$;
    $ELNU(Xy).u$ += $E_{new}.u$;
    $ELNU(Xy).pub$ += $E_{new}.pub$;
    i++; j++;
  else if $E_x.tid < E_y.tid$ **then** i++ ;
  else j++ ;
return $ELNU(Xy)$;

4.5. Correctness and Completeness

The MHEINU algorithm accurately and completely identifies all *HEIs* in a dataset containing negative utilities, based on a user-defined threshold, as mentioned below.

The algorithm represents all possible itemsets in the given dataset as an enumeration tree, organized according to the total processing order of the items (as illustrated in Figure 3). It then traverses the itemsets using a depth-first search strategy. During this traversal, the algorithm applies two pruning strategies, Pruning Strategies 1 and 2, to eliminate unpromising itemsets from the search space.

Pruning Strategy 1 uses the *uben* values of itemsets. Theorems 3 and 4 ensure that if an itemset's *uben* value is below the user-defined threshold, neither the itemset nor any of its supersets can be a *HEI*. Pruning Strategy 2 relies on the *ubeni* values of itemsets. Theorems 5 and 6 confirm that if an itemset's *ubeni* value is below the threshold, neither the itemset nor any of its extensions can be a *HEI*. Consequently, both pruning strategies help reduce the search space without the risk of missing any valid *HEIs*. The remaining unpruned itemsets are further analyzed by the MHEINU algorithm using their *ELNU* data structures. The *ELNU* of each itemset stores crucial utility and investment information, allowing the algorithm to evaluate the efficiency of each itemset. This enables the MHEINU algorithm to accurately determine whether the remaining itemsets qualify as *HEIs*, ensuring that no potential *HEI* is overlooked.

In conclusion, the MHEINU algorithm guarantees correctness by pruning itemsets that cannot possibly be *HEIs*, and by evaluating the remaining itemsets using their *ELNU* data structures. It ensures completeness by thoroughly assessing all unpruned itemsets, ensuring that no *HEI* is missed.

4.6. An Illustrated Example

This section presents the execution trace of the MHEINU algorithm using the sample database provided in Table 1, with *minE* set to 0.35.

The *ELNU* of each item appearing in the *DB* is constructed as shown in Figure 1. The *uben* values of items are provided in Table 7. When the items are sorted in ascending order based on their *uben* values, the resulting processing order $\prec$ is $c \prec g \prec b \prec e \prec d \prec f \prec a$. This ordering leads to the set-enumeration tree of the search space, as depicted in Figure 3.

Therefore, the algorithm begins by exploring the search space with item c. Since $uben(c) = 0.3462$, which is less than 0.35, item c or any of its supersets cannot be a HEI. Consequently, the algorithm continues with the next item, item g. Since $uben(g) = 0.4385$, which satisfies the threshold of 0.35, item g and its supersets will be examined. However, item g is not a HEI because its efficiency value $e(g) = 20/130 = 0.1538$, which is less than 0.35. The items that can be used to extend g include the items b, e, d, f, and a, respectively. Starting with item b, it is found that $ubeni(g, b) = 0.0576$, which is less than 0.35, leading to the pruning of this itemset. Similarly, the itemsets $\{g, e\}$ and $\{g, d\}$ are also pruned, as $ubeni(g, e) = 0.2809$ and $ubeni(g, d) = 0.3167$. The next item to be considered is item f. Since $ubeni(g, f) = 0.3608$, which meets the threshold of 0.35, the algorithm constructs the $ELNU$ for the itemset $\{g, f\}$. The constructed $ELNU(\{g, f\})$ is illustrated in Figure 4a. However, itemset $\{g, f\}$ or any of its extensions cannot be a HEI, since $uben(\{g, f\}) = 0.2342$, which is less than 0.35. The last item that can be used to extend g is item a. Since $ubeni(g, a) = 0.3701$, which exceeds the threshold of 0.35, the $ELNU$ for the itemset $\{g, a\}$ is constructed. The constructed $ELNU(\{g, a\})$ is shown in Figure 4b. However, $uben(\{g, a\}) = 0.2532$, which is less than 0.35. Therefore, the search space examination with item g is completed.

Next, the algorithm moves to item b. As $uben(b) = 0.8889$, which is greater than 0.35, item b and its extensions require further exploration. Item b is classified as a HEI due to its efficiency value $e(b) = 6/9 = 0.6667$, which exceeds the threshold of 0.35. However, all single-item extensions of b are pruned from the search space, as their $ubeni$ values are below 0.35: $ubeni(b, e) = 0.1404$, $ubeni(b, d) = 0.1356$, $ubeni(b, f) = 0.2162$, and $ubeni(b, a) = 0.2424$.

The algorithm continues to explore the search space by performing the same steps for the remaining items e, d, f, and a. Figure 4 illustrates the constructed $ELNUs$ of the visited itemsets, while Figure 5 presents the visited and pruned itemsets by the MHEINU algorithm during the search process.

Tid	*u*	*pub*
3	1	24
8	1	13
u : 2		
pub: 37		
inv : 158		

(**a**) $ELNU(\{g, f\})$

Tid	*u*	*pub*
3	5	25
8	4	14
u : 9		
pub: 39		
inv : 154		

(**b**) $ELNU(\{g, a\})$

Tid	*u*	*pub*
3	18	23
5	13	15
8	8	12
u : 39		
pub: 50		
inv : 98		

(**c**) $ELNU(\{e, d\})$

Tid	*u*	*pub*
3	17	23
5	11	15
8	7	12
u : 35		
pub: 50		
inv : 126		

(**d**) $ELNU(\{e, d, f\})$

Tid	*u*	*pub*
3	21	23
5	15	15
8	10	12
u : 46		
pub: 50		
inv : 122		

(**e**) $ELNU(\{e, d, a\})$

Tid	*u*	*pub*
3	−3	23
5	−4	15
7	−3	0
8	−3	12
u : −13		
pub: 50		
inv : 76		

(**f**) $ELNU(\{e, f\})$

Tid	*u*	*pub*
3	0	23
5	−2	15
8	−1	12
u : −3		
pub: 50		
inv : 100		

(**g**) $ELNU(\{e, f, a\})$

Tid	*u*	*pub*
3	1	23
5	0	15
8	0	12
u : 1		
pub: 50		
inv : 72		

(**h**) $ELNU(\{e, a\})$

Tid	*u*	*pub*
1	3	8
3	19	24
5	13	15
8	9	13
u : 44		
pub: 60		
inv : 78		

(**i**) $ELNU(\{d, f\})$

Tid	*u*	*pub*
1	8	8
3	22	24
5	15	15
8	11	13
u : 56		
pub: 60		
inv : 102		

(**j**) $ELNU(\{d, f, a\})$

Tid	*u*	*pub*
1	10	10
3	23	25
5	17	17
8	12	14
u : 62		
pub: 66		
inv : 74		

(**k**) $ELNU(\{d, a\})$

Tid	*u*	*pub*
1	3	8
3	2	24
5	0	15
8	1	13
u : 6		
pub: 60		
inv : 52		

(**l**) $ELNU(\{f, a\})$

Figure 4. The constructed $ELNUs$ of itemsets for the running example.

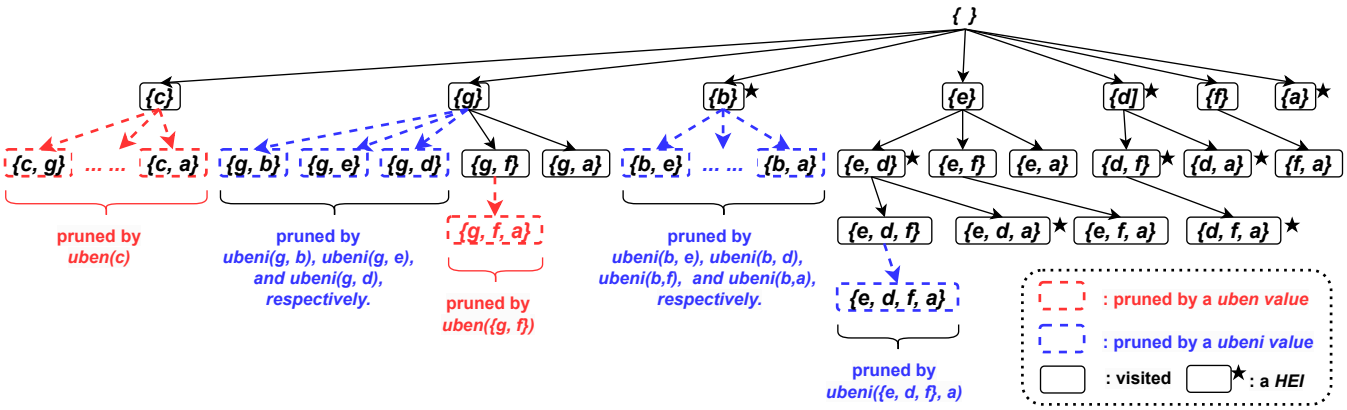


Figure 5. The visited itemset for the running example.

5. Experimental Analysis

This section presents the performance evaluation of the proposed MHEINU algorithm. Based on the available literature, no prior work has addressed the HEIM problem with negative utilities. Therefore, to individually assess the effectiveness of the designed pruning strategies and the total processing order among items, two additional algorithms, named MHEINU_woPS2 and MHEINU_lex, were also implemented and used in the performance evaluation. The MHEINU_woPS2 algorithm, differently from MHEINU, lacks a Pruning Strategy 2; that is, it does not check the condition in line 2 of the Search algorithm (Algorithm 2). On the other hand, the difference between MHEINU_lex and MHEINU is that MHEINU_lex takes into account the lexicographic order of items, meaning it sorts them alphabetically when executing line 2 of the MHEINU algorithm (Algorithm 1). Table 8 summarizes the properties of the algorithms that were compared. The algorithms were implemented in the Java programming language, and all experiments were conducted on the same computer running Windows 10, equipped with an i5-5200U 2.2 GHz processor and 8 GB of RAM.

Table 8. Compared algorithms.

Algorithm	Processing Order	Pruning Strategies
MHEINU_woPS2	*uben*-ascending	Pruning Strategy 1
MHEINU_lex	lexicographic	Pruning Strategies 1 and 2
MHEINU	*uben*-ascending	Pruning Strategies 1 and 2

The algorithms were compared in terms of runtime, memory consumption, the total number of join operations, and scalability. For the comparison, six datasets with varying characteristics were obtained from the open-source data mining library, SPMF [47]. Originally, these datasets did not contain the investment values, while providing the itemset utilities. The investment values for the items were generated as outlined in [23]. Accordingly, the total investment value for each item was randomly generated using a Gaussian distribution with $N(10{,}000, 10^2)$. If the generated value was less than zero, a smaller positive value was generated instead, using a Gaussian distribution with $N(100, 5^2)$. Once the investment values had been determined, they were added to the datasets as new lines following the transactions. The characteristics of these datasets are provided in Table 9, where $|T|$ is the total number of transactions, $|PI|$ is the total number of positive items, $|NI|$ is the total number of negative items, $AvgL$ is the average length of transactions, and $Density$ is calculated as $AvgL/(|PI| + |NI|) \times 100$.

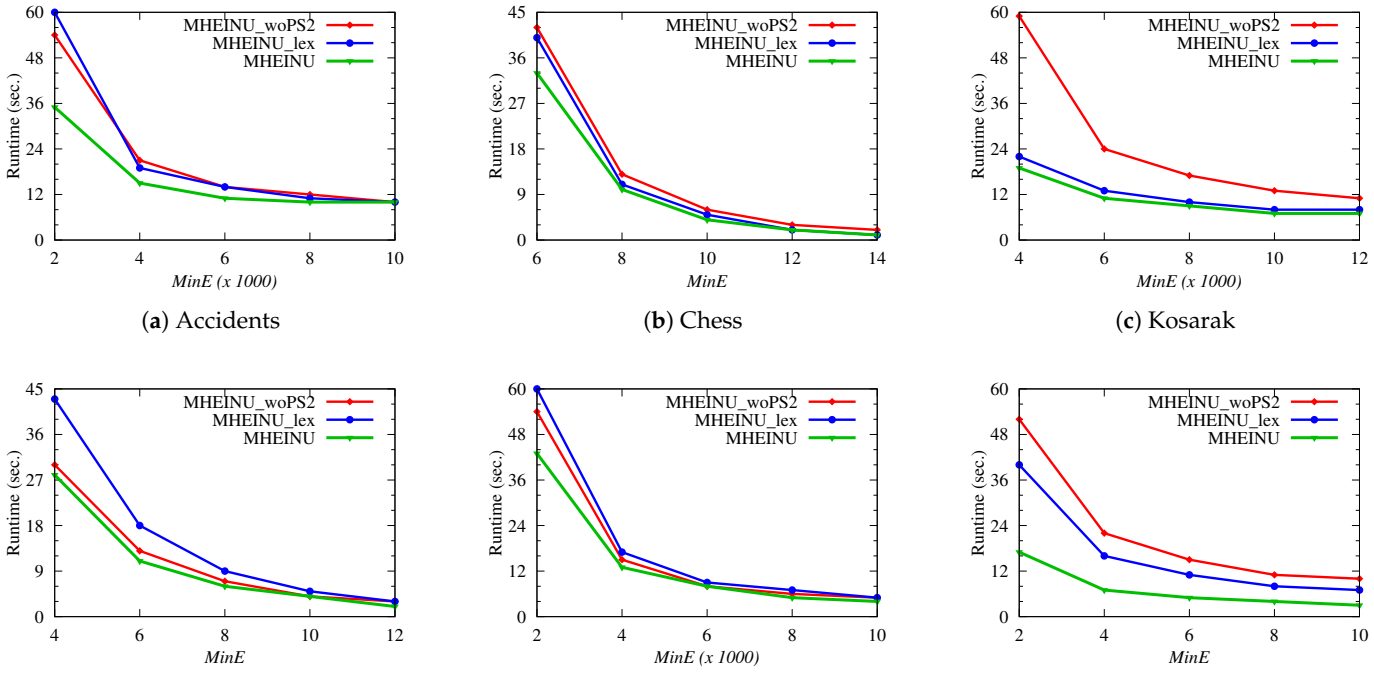
Table 9. Experimental datasets.

Dataset	|*T*|	|*PI*|	|*NI*|	*AvgL*	*Density*
Accidents	340,183	230	238	33.8	7.22
Chess	3196	37	38	37	49.33
Kosarak	990,002	20,700	20570	8.1	0.02
Mushroom	8124	74	45	23	19.33
Pumsb	49,046	1079	1034	74	3.50
Retail	88,162	8223	8247	10.3	0.06

5.1. Runtime

In the experiment, the runtime performance of the MHEINU algorithm was analyzed and compared with MHEINU_woPS2 and MHEINU_lex. All algorithms were executed on each experimental dataset under various *MinE* thresholds. Figure 6 presents the runtime results, illustrating the performance of each algorithm across the various datasets under different thresholds.

As can be seen in Figure 6, the runtime of each algorithm increased when the *MinE* value for each dataset was decreased. This is reasonable, because lower *MinE* values lead to an increase in the number of *HEIs*, which requires more itemsets to be examined in the search space. It was also observed that MHEINU consistently outperformed both MHEINU_woPS2 and MHEINU_lex across all experiments. Additionally, it was seen that, as the *MinE* value decreased, the runtime differences between MHEINU and MHEINU_woPS2, as well as between MHEINU and MHEINU_lex, became more pronounced. These results aligned with expectations, and the reasons for this are explained as follows.



(**a**) Accidents (**b**) Chess (**c**) Kosarak

(**d**) Mushroom (**e**) Pumsb (**f**) Retail

Figure 6. Runtime.

The reason MHEINU was faster than MHEINU_woPS2 is that MHEINU utilizes both Pruning Strategy 1 and Pruning Strategy 2 to reduce the search space, whereas MHEINU_woPS2 only employs Pruning Strategy 1. Pruning Strategy 2 uses the tighter upper-bound *ubeni*, which eliminates more unpromising itemsets compared to the looser upper-bound *uben* used in Pruning Strategy 1. As a result, MHEINU prunes the search

space more efficiently, examines fewer itemsets, and thus runs faster. The increase in the runtime difference between these two algorithms as the threshold *MinE* decreased was due to the increase in the number of *HEIs* that needed to be explored, i.e., the larger the search space. As the value of *MinE* decreases, the pruning efficiency of Pruning Strategy 1 decreases further, due to the looser upper-bound *uben* it uses. In contrast, Pruning Strategy 2, which uses *ubeni*, a tighter upper-bound compared to *uben*, can prune some itemsets that Pruning Strategy 1 cannot, thus contributing to the faster performance of MHEINU. For example, for the Accidents dataset, the difference in runtime between the two algorithms at the largest *MinE* (threshold of 10,000) was only about 0.4 s, while at the smallest *MinE* (threshold of 2000), this difference increased significantly to about 18 s. Similar results were observed for the Chess, Mushroom, Kosarak, Pumsb, and Retail datasets: at the largest *MinE* values, the runtime differences were around 0.7 s for Chess, 5 s for Kosarak, 0.4 s for Mushroom, 0.3 s for Pumsb, and 6 s for Retail. Meanwhile, at the smallest *MinE* values, these differences increased to approximately 10 s for Chess, 40 s for Kosarak, 2.5 s for Mushroom, 10 s for Pumsb, and 34 s for Retail.

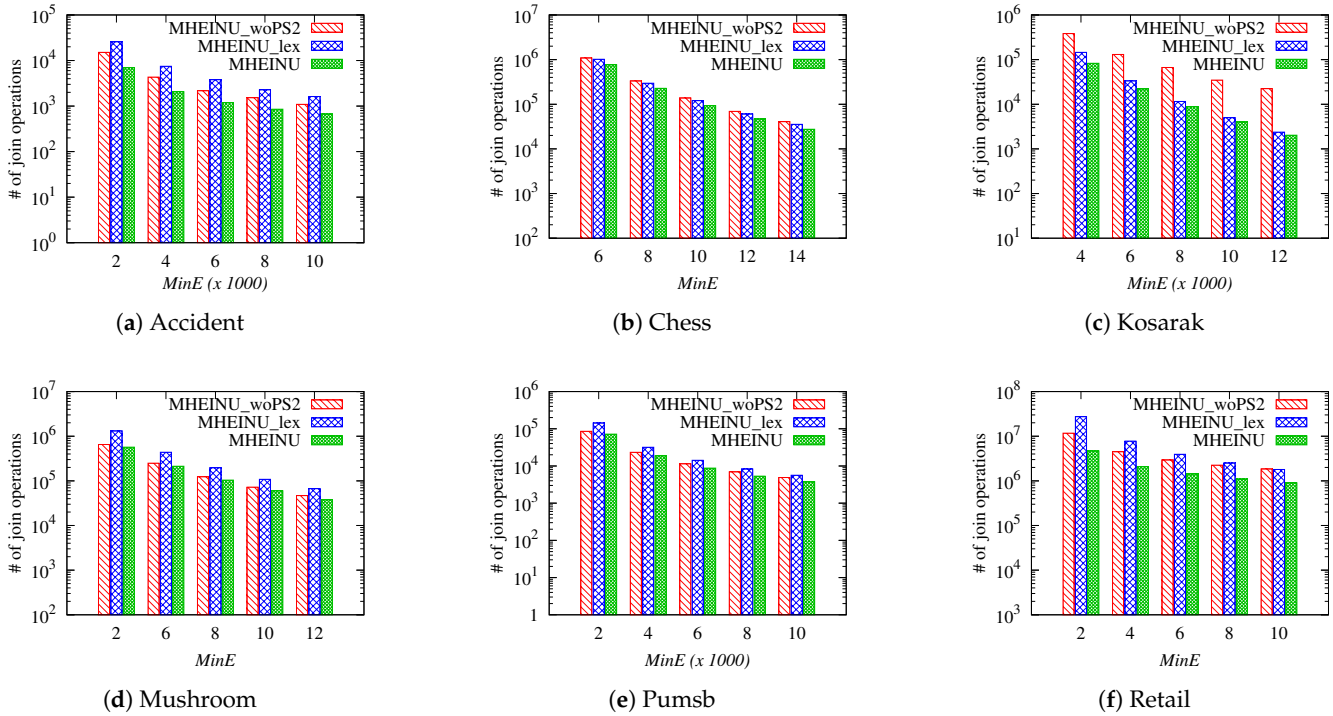
The reason MHEINU performed faster than MHEINU_lex lies in its use of the *uben*-ascending order when examining the search space, whereas MHEINU_lex relies on alphabetical ordering. This is reasonable, since processing items in *uben*-ascending order allows items with lower *uben* values and their extensions to be examined first, enabling earlier identification of unpromising itemsets and more efficient pruning of the search space. Consequently, although both algorithms employ the same pruning strategies, the *uben*-ascending order gives MHEINU a significant runtime advantage. This advantage becomes even more pronounced as the *MinE* threshold decreases, due to the expansion of the search space and the increased need for effective pruning. For example, at the largest *MinE* setting, MHEINU was faster than MHEINU_lex by 0.3, 0.4, 1, 0.9, 0.7, and 3.8 s on the Accidents, Chess, Kosarak, Mushroom, Pumsb, and Retail datasets, respectively. However, at the smallest *MinE* setting, the difference increased rapidly to 24, 7, 3, 40, 15, 16, and 23 s, respectively.

When comparing MHEINU_woPS2 and MHEINU_lex, it is observed that they exhibited a varying runtime superiority across the different experiments. For example, MHEINU_woPS2 outperformed MHEINU_lex on the Mushroom and Pumsb datasets, while MHEINU_lex performed better on the Chess, Kosarak, and Retail datasets. Additionally, on the Accidents dataset, MHEINU_woPS2 outperformed MHEINU_lex at *MinE* = 2000, while MHEINU_lex excelled at other settings. Overall, these results indicate that the characteristics of the datasets affected the runtime performance of both algorithms. However, despite lacking a Pruning Strategy 2, the fact that MHEINU_woPS2 outperformed MHEINU_lex in some experiments further emphasizes the contribution of considering the *uben*-ascending order of items to the runtime performance of the proposed MHEINU algorithm.

In summary, the experimental results demonstrated that incorporating both Pruning Strategy 1 and Pruning Strategy 2, along with structuring the search space based on the *uben*-ascending values of the items, significantly enhanced the efficiency of the proposed MHEINU algorithm, regardless of the size or density of the datasets.

5.2. Number of Join Operations

In this experiment, the number of join operations performed by the algorithms was analyzed to further understand their runtime performance. A join operation refers to invoking the Construct Algorithm to extend an itemset in the search space. Figure 7 presents the results for the total number of join operations performed by the algorithms for each dataset and threshold.



(**a**) Accident (**b**) Chess (**c**) Kosarak

(**d**) Mushroom (**e**) Pumsb (**f**) Retail

Figure 7. Number (#) of join operations.

The results, as expected, demonstrate that the MHEINU algorithm performed significantly fewer join operations compared to both MHEINU_woPS2 and MHEINU_lex across all datasets. The reason MHEINU performed fewer join operations than MHEINU_woPS2 lies in the absence of Pruning Strategy 2 in MHEINU_woPS2. By leveraging Pruning Strategy 2, MHEINU prunes the search space more effectively than MHEINU_woPS2, thereby reducing the number of join operations required during the mining process. The reason MHEINU performed fewer join operations than MHEINU_lex is its use of *uben*-ascending order in processing items. Processing items in *uben*-ascending order enables earlier pruning of unpromising itemsets, resulting in a more efficient mining process with fewer join operations.

For example, for the Accidents dataset, MHEINU achieved approximately 1.63 times fewer join operations than MHEINU_woPS2 and about 2.41 times fewer than MHEINU_lex when $minE$ = 10,000. This trend persisted at lower thresholds; when $MinE$ 8000, MHEINU performed approximately 1.79 times fewer join operations than MHEINU_woPS2 and about 2.67 times fewer than MHEINU_lex. When $MinE$ = 6000, MHEINU again showed a notable reduction, requiring roughly 1.81 times fewer join operations than MHEINU_woPS2 and about 3.18 times fewer than MHEINU_lex. When $MinE$ 4000, MHEINU executed approximately 2.09 times fewer join operations than MHEINU_woPS2 and about 3.61 times fewer than MHEINU_lex. Finally, at the lowest threshold ($MinE$ = 2000), MHEINU performed around 2.16 times fewer join operations than MHEINU_woPS2 and about 3.71 times fewer than MHEINU_lex. Comparisons with respect to the smallest threshold value on other datasets were as follows. On the Chess dataset, when $MinE$ = 6, MHEINU performed approximately 1.42 times fewer join operations than MHEINU_woPS2 and approximately 1.32 times fewer join operations than MHEINU_lex. On the Kosarak dataset, when $MinE$ = 4000, MHEINU showed a significant reduction and required approximately 4.63 times fewer join operations compared to MHEINU_woPS2 and approximately 1.75 times fewer join operations than MHEINU_lex. On the Mushrooms dataset, when $MinE$ = 2, MHEINU performed approximately 1.14 times fewer join operations than

MHEINU_woPS2 and about 2.30 times fewer join operations than MHEINU_lex. On the Pumsb dataset, when $MinE$ = 2000, MHEINU required approximately 1.19 times fewer join operations than MHEINU_woPS2 and about 1.98 times fewer join operations than MHEINU_lex. Finally, for the Retail dataset, when $MinE$ = 2, MHEINU required approximately 2.49 times fewer join operations than MHEINU_woPS2 and about 5.84 times fewer join operations than MHEINU_lex.

As a result, these findings emphasize the effectiveness of the proposed MHEINU algorithm in pruning the search space and explain its superior performance compared to other algorithms in terms of runtime. It can be concluded that both the use of pruning strategies and the exploration of the search space based on the *uben*-ascending order of items individually contribute to effectively reducing the computational overhead by minimizing the number of necessary join operations.

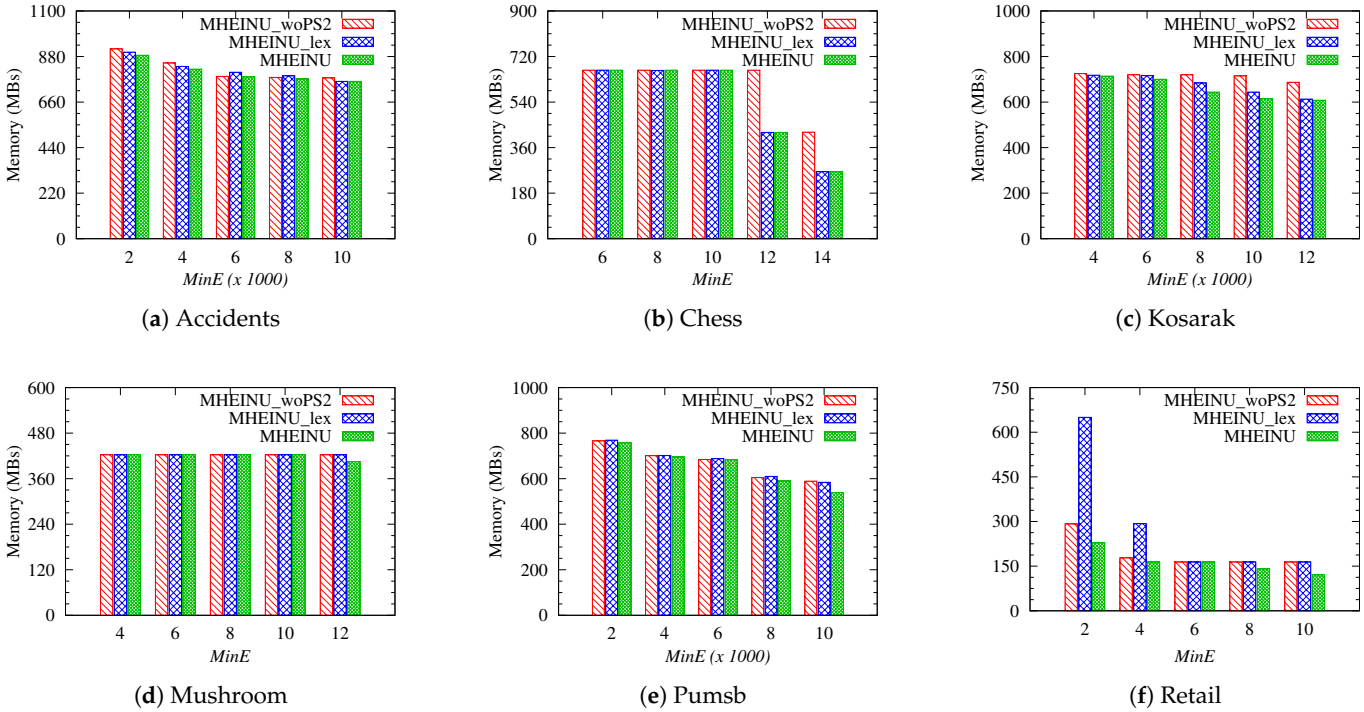
On the other hand, when comparing MHEINU_woPS2 and MHEINU_lex algorithms, it was observed that the runtime and the number of join operations results were consistent for all datasets, except the experiments performed on the Accident dataset with the settings $minE \geq 4000$, and Retail. Interestingly, in these experiments, MHEINU_lex performed more join operations than MHEINU_woPS2 but was more efficient in terms of runtime. Further analysis of the results revealed that this discrepancy was due to MHEINU_lex performing merge operations with relatively smaller $ELNUs$ compared to MHEINU_woPS2. This makes sense, because joining larger $ELNUs$ naturally takes more time than joining smaller ones, which explains why MHEINU_lex, despite performing more join operations in some experiments, completed the mining process faster than MHEINU_woPS2. As a result, this observation is quite valuable, as it demonstrates that items with lower *uben* values may not always have smaller $ELNUs$. This finding will serve as an important guide for future studies aiming to solve the problem more effectively, as it highlights the need to investigate different processing orders among items, depending on the characteristics of the dataset. However, it is important to note that the MHEINU algorithm exhibited the best performance in these experiments as well, both in terms of runtime and the number of join operations.

In summary, the reason for MHEINU's fast performance is that it uses Pruning Strategies 1 and 2 together, taking into account the *uben*-ascending order of items. In this way, it performs fewer join operations and completes the join operations it performs in a more efficient manner.

5.3. Memory

In this experiment, the memory consumption of the algorithms was analyzed. Figure 8 presents the memory consumption results, illustrating the performance of each algorithm across various datasets and under different threshold values.

The results show that the memory usage of each algorithm increased as $minE$ decreased. This can be explained by the larger search space at lower $minE$ values. As the search space grows, the algorithms visit more itemsets, which means the number of stored $ELNUs$ increases. It was also observed that the memory consumption of the algorithms in all experiments was closely related to the number of join operations they performed. This is reasonable, because a higher number of join operations typically means that algorithms explore deeper levels of the search space, i.e., processing larger itemsets. Therefore, more $ELNUs$ need to be stored in memory to construct $ELNUs$ for larger itemsets. Therefore, an increase in the number of join operations is directly related to an increase in the amount of memory consumed.



(**a**) Accidents (**b**) Chess (**c**) Kosarak

(**d**) Mushroom (**e**) Pumsb (**f**) Retail

Figure 8. Memory.

Furthermore, as can be seen in Figure 8, the proposed MHEINU algorithm had a lower memory usage compared to the others in all experiments. This difference became especially pronounced at higher threshold values. At lower threshold values, the decreasing differences in memory usage between the algorithms can be attributed to the expansion of the search space. In larger search spaces, the algorithms tended to explore to similar depths, due to the increasing length of the itemsets to be explored, thus decreasing the differences in memory usage.

For example, in the experiments conducted on the Accidents dataset, MHEINU used up to 32 MB less memory compared to MHEINU_woPS2 and up to 15 MB less than MHEINU_lex. On the Chess dataset, MHEINU provided a memory saving of up to 156 MB compared to MHEINU_woPS2, with almost no difference observed when compared to MHEINU_lex. On the Kosarak dataset, MHEINU reduced the memory usage by up to 100 MB compared to MHEINU_woPS2 and by 40 MB compared to MHEINU_lex. For the Mushroom dataset, MHEINU achieved up to 19 MB of memory savings relative to both other algorithms. On the Pumsb dataset, MHEINU used up to 50 MB less memory than MHEINU_woPS2 and 45 MB less than MHEINU_lex. Lastly, on the Retail dataset, MHEINU achieved a memory reduction of up to 23 MB compared to MHEINU_woPS2 and a significant reduction of up to 421 MB compared to MHEINU_lex.

In conclusion, the proposed MHEINU algorithm consistently exhibited a lower memory usage compared to the other algorithms, regardless of the varying characteristics of the datasets. This highlights the effectiveness of the MHEINU algorithm in handling memory usage under varying dataset and *minE* settings.

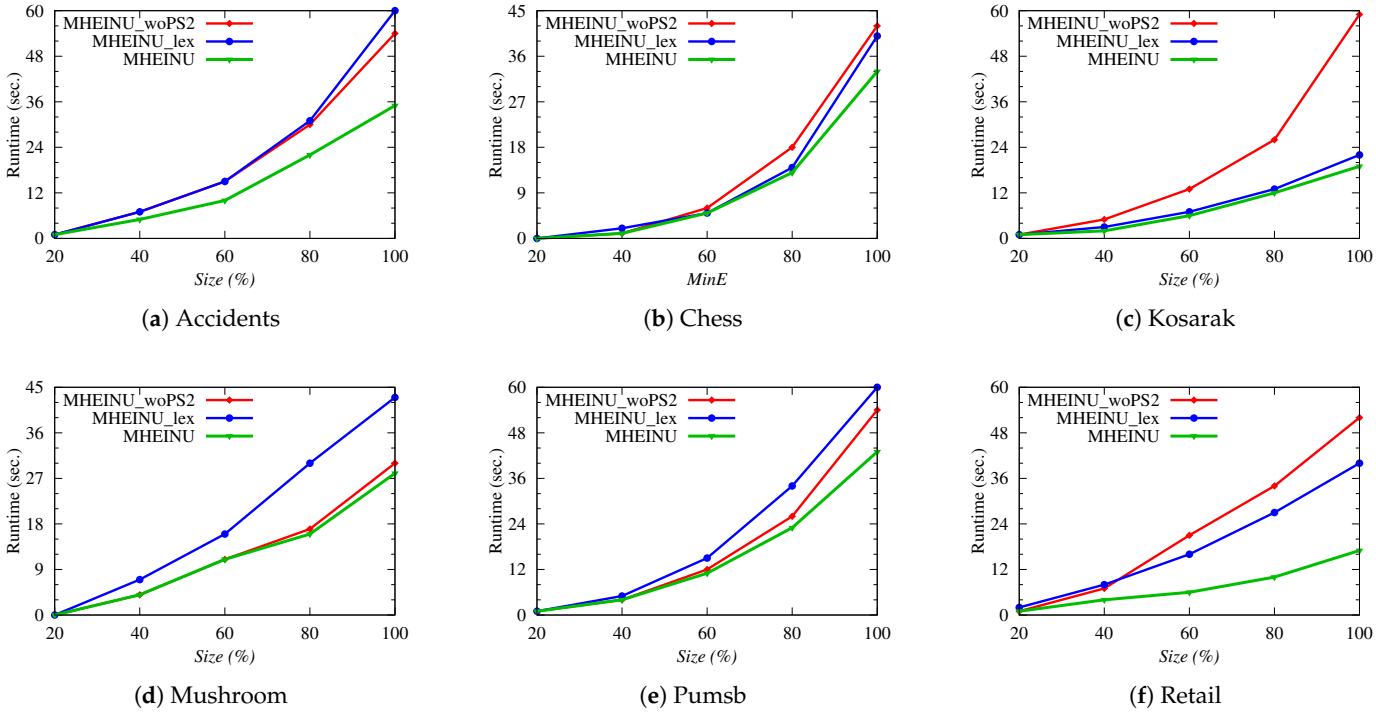
5.4. Scalability

In this section, the scalability performance of the algorithms was analyzed. To perform this analysis, several datasets were generated by resizing each experimental dataset to include the first X% transactions, where X% varied between 20% and 100%. The experi-

ments were performed on the same datasets of varying sizes, using a fixed *MinE* value (the smallest *MinE* used in previous experiments). The results are presented in Figure 9.

As seen in Figure 9, the algorithms showed an increase in runtime as the dataset size increased. This is reasonable, because the *MinE* value remains constant as the dataset size increases, and it is expected that larger datasets will take more time to process. However, the runtime increases showed a nearly linear trend for each algorithm. Therefore, it can be said that all algorithms exhibited good scalability in terms of runtime. In addition, the MHEINU algorithm had a better performance than both MHEINU_woPS2 and MHEINU_lex in all experiments. Furthermore, it was observed that, as the dataset size increased, the runtime difference between MHEINU and the other algorithms became more pronounced.

For example, for the Accidents dataset, when considering 20% of the dataset, the algorithms exhibited nearly identical runtimes. However, as the dataset size increased, MHEINU demonstrated a growing performance advantage. Specifically, for the Accidents dataset at 20%, 40%, 60%, 80%, and 100% sizes, MHEINU ran 0.8 and 0.13 s faster than MHEINU_woPS2 and MHEINU_lex at 20%, 2.29 and 1.8 s faster at 40%, 5.14 and 5.32 s faster at 60%, 8.59 and 8.93 s faster at 80%, and 18.41 and 24.21 s faster at 100%. Similarly, for the other datasets, while the algorithms had similar runtimes at the lowest dataset sizes, MHEINU's superiority became increasingly evident as the size grew. At the 100% dataset size, the results for the other datasets were as follows. For the Chess dataset, MHEINU outperformed MHEINU_woPS2 by 9.84 s and MHEINU_lex by 7.1 s. On the Kosarak dataset, MHEINU was 40.1 s faster than MHEINU_woPS2 and 3.02 s faster than MHEINU_lex. In the case of the Mushroom dataset, MHEINU ran 4.2 s faster than MHEINU_woPS2 and 15.6 s faster than MHEINU_lex. For the Pumsb dataset, MHEINU was 10.55 s faster than MHEINU_woPS2 and 16.28 s faster than MHEINU_lex. Finally, for the Retail dataset, MHEINU exhibited a remarkable runtime advantage, being 34.37 s faster than MHEINU_woPS2 and 22.3 s faster than MHEINU_lex. In summary, the results highlight that MHEINU exhibited better scalability compared to the other algorithms as the dataset size increases.



(**a**) Accidents (**b**) Chess (**c**) Kosarak

(**d**) Mushroom (**e**) Pumsb (**f**) Retail

Figure 9. Scalability.

5.5. Number of Discovered Itemsets

In this experiment, to more clearly demonstrate the necessity of developing new techniques and methods for solving the HEIM problem in datasets containing negative items, the number of *HEIs* discovered by the proposed MHEINU algorithm was compared with those discovered by the HEPMiner [23] and MHEI [24] algorithms, which were designed to address the classical HEIM problem. The algorithms were run on experimental datasets using the same *minE* settings as in the previous experiments, and the results are presented in Table 10.

Table 10. Comparison of the number of *HEIs* discovered by MHEINU, HEPMiner, and MHEI.

(a) Accidents				
		# of *HEIs* discovered by		
minE	**# of *HEIs***	**MHEINU**	**HEPMiner**	**MHEI**
10,000	6	6	2	5
8000	9	9	9	8
6000	16	16	6	7
4000	25	25	9	13
2000	56	56	10	12
(b) Chess				
		# of *HEIs* discovered by		
minE	**# of *HEIs***	**MHEINU**	**HEPMiner**	**MHEI**
14	607	607	10	152
12	673	673	11	169
10	777	777	12	210
8	932	932	12	231
6	1226	1226	12	255
(c) Kosarak				
		# of *HEIs* discovered by		
minE	**# of *HEIs***	**MHEINU**	**HEPMiner**	**MHEI**
12,000	5	5	5	5
10,000	5	5	5	5
8000	5	5	5	5
6000	5	5	5	5
4000	10	10	9	10
(d) Mushroom				
		# of *HEIs* discovered by		
minE	**# of *HEIs***	**MHEINU**	**HEPMiner**	**MHEI**
12	784	784	564	612
10	1003	1003	725	779
8	1409	1409	1022	1054
6	2366	2366	1871	1666
4	5919	5919	4666	3904
(e) Pumsb				
		# of *HEIs* discovered by		
minE	**# of *HEIs***	**MHEINU**	**HEPMiner**	**MHEI**
10,000	2	2	2	2
8000	8	8	8	7
6000	16	16	15	9
4000	42	42	42	18
2000	237	237	226	224

Table 10. *Cont.*

(f) Retail				
		# of *HEIs* discovered by		
minE	**# of *HEIs***	**MHEINU**	**HEPMiner**	**MHEI**
10	333	333	315	333
8	408	408	393	408
6	546	546	514	545
4	816	816	722	809
2	1644	1644	1312	1609

As expected, MHEINU discovered all *HEIs* across all datasets and *minE* thresholds, while HEPMiner and MHEI failed to discover some *HEIs*. This is because MHEINU was specifically designed for datasets with negative items, whereas HEPMiner and MHEI assume datasets contain only positive items, limiting their ability to find all *HEIs*. In other words, the inability of HEPMiner and MHEI to discover some *HEIs* in such datasets stems from their incorrect pruning of the search space.

The results show that the number of *HEIs* missed by HEPMiner and MHEI increased significantly as *minE* decreased. This is because, as *minE* lowers, the *HEIs* to be discovered become longer. The presence of negative items tended to cause the upper-bound models used by HEPMiner and MHEI to underestimate the values, especially for longer itemsets. As a result, as *minE* decreased, these algorithms pruned the search space more excessively, missing more *HEIs*. For example, based on the results of the experiments with the lowest *minE* settings, the proportions of *HEIs* missed by HEPMiner and MHEI were approximately as follows: Accident (82% and 78%), Ches (99% and 79%), Mushroom (21% and 34%), Pumsb (5% and 6%), and Retail (20% and 2%). Note that, in the experiments conducted on the Kosarak dataset, it was observed that HEPMiner and MHEI successfully discovered all the *HEIs* (except for HEPMiner when $minE = 4000$). This can be explained by the fact that the Kosarak dataset is large and sparse and the *minE* settings used in the experiments were not low enough for Kosarak, resulting in a relatively small search space. This is reasonable, because in experiments conducted on Kosarak using lower *minE* settings (e.g., when $minE \leq 2500$), it was observed that the number of *HEIs* missed by these algorithms increased.

On the other hand, the performance of the HEPMiner and MHEI algorithms differed in terms of missed *HEI* across the different datasets and *minE* values. This is reasonable, since they used different upper-bound models to prune the search space. Therefore, they may exhibit different behaviors in over-pruning the search space, depending on the characteristics of the datasets, such as their size, density, and the number of negative items they contain.

Consequently, the existing HEIM algorithms do not guarantee the discovery of all *HEIs* when databases contain negative items. This highlights the importance of the proposed MHEINU algorithm in handling the complete discovery of *HEIs* from datasets with negative items (utilities).

6. Limitations of HEIM

Although the proposed MHEINU algorithm overcomes the issue of the incomplete discovery of *HEIs* in datasets with negative utilities, HEIM still has some limitations for certain applications. This section addresses these limitations and provides an overview of potential extensions that could enhance its practical impact by meeting the needs of various applications.

One limitation occurs when the *minE* threshold is set too high or too low. When *minE* is set too high, a relatively small number of *HEIs* may be discovered, preventing important itemsets from being presented to the user. On the other hand, when *minE* is set too low, a large number of *HEIs* may be found, making the analysis process time-consuming for decision-makers. Determining an appropriate *minE* threshold can be challenging for users. To overcome this problem, the HEIM problem could be transformed into finding the *k* most efficient itemsets, with the user specifying the value of *k* instead of *minE*. The literature has discussed finding the *k* most important itemsets in earlier itemset problems [48], and similar approaches could be adapted to HEIM. For instance, by initially setting *minE* to 0, the search space is traversed. As soon as the first *k* high-efficiency itemsets are found, *minE* is updated to match the efficiency of the lowest efficiency itemset among them. The search then continues, with the *k* most efficient itemsets and *minE* being dynamically updated as new *HEIs* are discovered.

Another challenge arises when many of the discovered *HEIs* contain weakly correlated items. Such itemsets can be misleading for decision-making, especially for campaign planning, because marketing strategies involving weakly correlated items are unlikely to be effective. In the literature, there are measurements that assess how closely the items in itemsets are related to each other. Addressing this issue, it would be valuable to extend the HEIM problem by incorporating techniques [49] that measure the strength of item correlations and include these relationships to generate more meaningful results.

Additionally, HEIM does not account for the time information of transactions, treating all transactions equally, regardless of whether they are recent or older. However, items from more recent transactions might be more relevant or significant, due to factors such as seasonal trends, upcoming holidays, and specific events. Therefore, it is essential to extend the problem to include the discovery of temporal *HEIs* that consider these time-sensitive factors. One way to address this is by adapting the HEIM problem to handle temporal data using techniques such as the sliding window or damped window approach. Using a sliding window technique, the focus may be on discovering *HEIs* within a specific, fixed time window, which allows the analysis to capture trends over a defined period. On the other hand, by using the damped window technique, more weight can be assigned to new transactions, thus allowing the recent data to have a greater impact on the discovery of *HEIs*.

Last but not least, another limitation of this study lies in the assumptions made regarding fixed investment values. In many real-world applications, the value of an investment may fluctuate over time due to market conditions, changes in demand, or other external factors. However, the current study assumed that investment values remained fixed, which may not accurately reflect the dynamic nature of many business environments. This assumption may limit the practical applicability of the study in scenarios where investments change frequently or where real-time adjustments are required. To overcome this, it is important to conduct new studies by incorporating variable investment values into HEIM, using dynamic pricing or adjusting investment values based on temporal data. This may enable HEIM to better capture the real-world complexity of changing investment landscapes and provide more robust and adaptable results for decision-makers.

7. Conclusions and Future Work

The existing algorithms proposed for the problem of HEIM were designed under the assumption that datasets contain only positive utilities. However, real-world datasets contains also negative utilities. As a result, existing HEIM algorithms encounter incomplete high-efficiency itemset discovery when a database also contains negative utilities. To address this issue, this study introduced a novel algorithm called MHEINU. MHEINU

utilizes two new upper-bounds to effectively and safely prune the search space, along with a list-based data structure designed to minimize the costs associated with database scans. Experimental results on various datasets containing negative utilities showed that MHEINU effectively discovered the complete set of high-efficiency itemsets. Furthermore, MHEINU performed efficiently in terms of runtime, number of join operations, and memory usage, and exhibited good scalability as datasets grew. In addition to its algorithmic advancements, MHEINU has significant potential for real-world applications. For example, in supply chain optimization, it could assist with inventory management by identifying product combinations that maximize profits with limited capital. Additionally, it could optimize customer satisfaction and sales traffic by enabling the development of bundled or cross-selling strategies, where some products are sold at a loss but still provide the targeted profit margin. Thus, it could also help uncover relationships between campaigns and customer purchasing behavior. These capabilities make MHEINU particularly valuable in scenarios involving complex datasets with positive and negative utilities. Furthermore, MHEINU's ability to analyze datasets with both positive and negative utilities may open up new opportunities in other areas, such as financial analysis. For example, it could reveal profitable yet low-risk investment strategies. In the future, studies could be conducted to design tighter upper-bounds and investigate more appropriate processing ordering among items, in order to further improve the solution efficiency of the HEIM problem. Additionally, it would also be interesting to investigate the effect of a suitable tree data structure on the problem-solving efficiency. Investigating domain-specific adaptations of MHEINU for supply chain, finance, and healthcare applications represents another promising direction for further research. Future work may also explore extending MHEINU to handle streaming data, where the algorithm could dynamically update high-efficiency itemsets as new data arrive in real-time. Since the utility, positive utility upper, and investment values of the items are stored in the *ELNU* data structure that MHEINU uses, MHEINU could be easily adapted to solve the problem of streaming data.

Funding: This research received no external funding.

Data Availability Statement: The datasets used in this study are available in SPMF [47]. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. Shaikh, M.; Akram, S.; Khan, J.; Khalid, S.; Lee, Y. DIAFM: An Improved and Novel Approach for Incremental Frequent Itemset Mining. *Mathematics* **2024**, *12*, 3930. [CrossRef]
2. Li, B.; Pei, Z.; Zhang, C.; Hao, F. Efficient Associate Rules Mining Based on Topology for Items of Transactional Data. *Mathematics* **2023**, *11*, 401. [CrossRef]
3. Csalódi, R.; Abonyi, J. Integrated Survival Analysis and Frequent Pattern Mining for Course Failure-Based Prediction of Student Dropout. *Mathematics* **2021**, *9*, 463. [CrossRef]
4. Zhao, X.; Zhang, X.; Wang, P.; Chen, S.; Sun, Z. A weighted frequent itemset mining algorithm for intelligent decision in smart systems. *IEEE Access* **2018**, *6*, 29271–29282. [CrossRef]
5. Chen, R.; Zhao, S.; Liu, M. A Fast Approach for Up-Scaling Frequent Itemsets. *IEEE Access* **2020**, *8*, 97141–97151. [CrossRef]
6. Sadeequllah, M.; Rauf, A.; Rehman, S.U.; Alnazzawi, N. Probabilistic Support Prediction: Fast Frequent Itemset Mining in Dense Data. *IEEE Access* **2024**, *12*, 39330–39350. [CrossRef]
7. Rai, S.; Kumar, P.; Shetty, K.N.; Geetha, M.; Giridhar, B. WBIN-Tree: A Single Scan Based Complete, Compact and Abstract Tree for Discovering Rare and Frequent Itemset Using Parallel Technique. *IEEE Access* **2024**, *12*, 6281–6297. [CrossRef]
8. Liu, Y.; Liao, W.-k.; Choudhary, A.A. A Two-Phase Algorithm for Fast Discovery of High Utility Itemsets. In *Advances in Knowledge Discovery and Data Mining*; Springer: Berlin/Heidelberg, Germany, 2005; pp. 689–695. [CrossRef]
9. Tseng, V.S.; Shie, B.E.; Wu, C.W.; Yu, P.S. Efficient Algorithms for Mining High Utility Itemsets from Transactional Databases. *IEEE Trans. Knowl. Data Eng.* **2013**, *25*, 1772–1786. [CrossRef]

10. Liu, M.; Qu, J. Mining high utility itemsets without candidate generation. In Proceedings of the 21st ACM International Conference on Information and Knowledge Management—CIKM, Maui, HI, USA, 29 October–2 November 2012; pp. 55–64. [CrossRef]
11. Fournier-Viger, P.; Wu, C.W.; Zida, S.; Tseng, V.S. FHM: Faster High-Utility Itemset Mining Using Estimated Utility Co-occurrence Pruning. In *Lecture Notes in Computer Science*; Springer International Publishing: Berlin/Heidelberg, Germany, 2014; pp. 83–92. [CrossRef]
12. Liu, J.; Wang, K.; Fung, B.C. Mining High Utility Patterns in One Phase without Generating Candidates. *IEEE Trans. Knowl. Data Eng.* **2016**, *28*, 1245–1257. [CrossRef]
13. Krishnamoorthy, S. Pruning strategies for mining high utility itemsets. *Expert Syst. Appl.* **2015**, *42*, 2371–2381. [CrossRef]
14. Zida, S.; Fournier-Viger, P.; Lin, J.C.W.; Wu, C.W.; Tseng, V.S. EFIM: A fast and memory efficient algorithm for high-utility itemset mining. *Knowl. Inf. Syst.* **2016**, *51*, 595–625. [CrossRef]
15. Ryang, H.; Yun, U. Indexed list-based high utility pattern mining with utility upper-bound reduction and pattern combination techniques. *Knowl. Inform. Syst.* **2016**, *51*, 627–659. [CrossRef]
16. Peng, A.Y.; Koh, Y.S.; Riddle, P. mHUIMiner: A Fast High Utility Itemset Mining Algorithm for Sparse Datasets. In *Advances in Knowledge Discovery and Data Mining*; Springer International Publishing: Berlin/Heidelberg, Germany, 2017; pp. 196–207. [CrossRef]
17. Krishnamoorthy, S. HMiner: Efficiently mining high utility itemsets. *Expert Syst. Appl.* **2017**, *90*, 168–183. [CrossRef]
18. Wu, P.; Niu, X.; Fournier-Viger, P.; Huang, C.; Wang, B. UBP-Miner: An efficient bit based high utility itemset mining algorithm. *Knowl. Based Syst.* **2022**, *248*, 108865. [CrossRef]
19. Nguyen, L.T.; Nguyen, P.; Nguyen, T.D.; Vo, B.; Fournier-Viger, P.; Tseng, V.S. Mining high-utility itemsets in dynamic profit databases. *Knowl. Based Syst.* **2019**, *175*, 130–144. [CrossRef]
20. Qu, J.F.; Fournier-Viger, P.; Liu, M.; Hang, B.; Hu, C. Mining High Utility Itemsets Using Prefix Trees and Utility Vectors. *IEEE Trans. Knowl. Data Eng.* **2023**, *35*, 10224–10236. [CrossRef]
21. Yan, Y.; Niu, X.; Zhang, Z.; Fournier-Viger, P.; Ye, L.; Min, F. Efficient high utility itemset mining without the join operation. *Inf. Sci.* **2024**, *681*, 121218. [CrossRef]
22. Liu, Y.; Wang, L.; Feng, L.; Jin, B. Mining High Utility Itemsets Based on Pattern Growth without Candidate Generation. *Mathematics* **2020**, *9*, 35. [CrossRef]
23. Zhang, X.; Chen, G.; Song, L.; Gan, W.; Song, Y. HEPM: High-efficiency pattern mining. *Knowl. Based Syst.* **2023**, *281*, 111068. [CrossRef]
24. Huynh, B.; Tung, N.; Nguyen, T.D.; Bui, Q.T.; Nguyen, L.T.; Yun, U.; Vo, B. An efficient strategy for mining high-efficiency itemsets in quantitative databases. *Knowl. Based Syst.* **2024**, *299*, 112035. [CrossRef]
25. Krishnamoorthy, S. Efficiently mining high utility itemsets with negative unit profits. *Knowl. Based Syst.* **2018**, *145*, 1–14. [CrossRef]
26. Singh, K.; Shakya, H.K.; Singh, A.; Bhaskar. Mining of high-utility itemsets with negative utility. *Expert Syst.* **2018**, *35*, e12296. [CrossRef]
27. Yildirim, I.; Celik, M. Mining high-average utility itemsets with positive and negative external utilities. *New Gener. Comput.* **2020**, *38*, 153–186. [CrossRef]
28. Zakaria, A.F.; Lim, S.C.J.; Aamir, M. A pricing optimization modelling for assisted decision making in telecommunication product-service bundling. *Int. J. Ing. Manag. Data Insights* **2024**, *4*, 100212. [CrossRef]
29. Singh, K.; Singh, S.S.; Kumar, A.; Biswas, B. High utility itemsets mining with negative utility value: A survey. *J. Intell. Fuzzy Syst.* **2018**, *35*, 6551–6562. [CrossRef]
30. Gan, W.; Lin, J.C.W.; Fournier-Viger, P.; Chao, H.C.; Tseng, V.S.; Yu, P.S. A Survey of Utility-Oriented Pattern Mining. *IEEE Trans. Knowl. Data Eng.* **2021**, *33*, 1306–1327. [CrossRef]
31. Baralis, E.; Cagliero, L.; Garza, P. Planning stock portfolios by means of weighted frequent itemsets. *Expert Syst. Appl.* **2017**, *86*, 1–17. [CrossRef]
32. Zhao, X.; Zhong, X.; Han, B. Frequent Closed High-Utility Itemset Mining Algorithm Based on Leiden Community Detection and Compact Genetic Algorithm. *IEEE Access* **2024**, *12*, 84763–84773. [CrossRef]
33. Xie, S.; Zhao, L. An Efficient Algorithm for Mining Stable Periodic High-Utility Sequential Patterns. *Symmetry* **2022**, *14*, 2032. [CrossRef]
34. Vu, V.V.; Lam, M.T.H.; Duong, T.T.M.; Manh, L.T.; Nguyen, T.T.T.; Nguyen, L.V.; Yun, U.; Snasel, V.; Vo, B. FTKHUIM: A Fast and Efficient Method for Mining Top-K High-Utility Itemsets. *IEEE Access* **2023**, *11*, 104789–104805. [CrossRef]
35. Lee, C.; Kim, H.; Cho, M.; Kim, H.; Vo, B.; Lin, J.C.W.; Fournier-Viger, P.; Yun, U. Incremental Top-k High Utility Pattern Mining and Analyzing Over the Entire Accumulated Dynamic Database. *IEEE Access* **2024**, *12*, 77605–77620. [CrossRef]
36. Vo, B.; Nguyen, L.V.; Vu, V.V.; Lam, M.T.H.; Duong, T.T.M.; Manh, L.T.; Nguyen, T.T.T.; Nguyen, L.T.T.; Hong, T.P. Mining Correlated High Utility Itemsets in One Phase. *IEEE Access* **2020**, *8*, 90465–90477. [CrossRef]

37. Oguz, D. Ignoring Internal Utilities in High-Utility Itemset Mining. *Symmetry* **2022**, *14*, 2339. [CrossRef]
38. Wu, J.M.T.; Lin, J.C.W.; Pirouz, M.; Fournier-Viger, P. TUB-HAUPM: Tighter Upper Bound for Mining High Average-Utility Patterns. *IEEE Access* **2018**, *6*, 18655–18669. [CrossRef]
39. Yildirim, I.; Celik, M. An Efficient Tree-Based Algorithm for Mining High Average-Utility Itemset. *IEEE Access* **2019**, *7*, 144245–144263. [CrossRef]
40. Kim, H.; Ryu, T.; Lee, C.; Kim, S.; Vo, B.; Lin, J.C.W.; Yun, U. Efficient Method for Mining High Utility Occupancy Patterns Based on Indexed List Structure. *IEEE Access* **2023**, *11*, 43140–43158. [CrossRef]
41. Duong, H.; Pham, H.; Truong, T.; Fournier-Viger, P. Efficient algorithms to mine concise representations of frequent high utility occupancy patterns. *Appl. Intell.* **2024**, *54*, 4012–4042. [CrossRef]
42. Tang, H.; Wang, J.; Wang, L. Mining Significant Utility Discriminative Patterns in Quantitative Databases. *Mathematics* **2023**, *11*, 950. [CrossRef]
43. Yildirim, I.; Celik, M. FIMHAUI: Fast Incremental Mining of High Average-Utility Itemsets. In Proceedings of the 2018 International Conference on Artificial Intelligence and Data Processing (IDAP), Malatya, Turkey, 28–30 September 2018; IEEE: Piscataway, NJ, USA, 2018; Volume 41, pp. 1–9. [CrossRef]
44. Sra, P.; Chand, S. A Reinduction-Based Approach for Efficient High Utility Itemset Mining from Incremental Datasets. *Data Sci. Eng.* **2023**, *9*, 73–87. [CrossRef]
45. Nam, H.; Yun, U.; Vo, B.; Truong, T.; Deng, Z.H.; Yoon, E. Efficient Approach for Damped Window-Based High Utility Pattern Mining With List Structure. *IEEE Access* **2020**, *8*, 50958–50968. [CrossRef]
46. Yildirim, I. Mining High Average-Efficiency Itemsets. In Proceedings of the 8th International Artificial Intelligence and Data Processing Symposium (IDAP), Malatya, Turkey, 21–22 September 2024; IEEE: Piscataway, NJ, USA, 2024; pp. 1–6. [CrossRef]
47. Fournier-Viger, P.; Lin, J.C.W.; Gomariz, A.; Gueniche, T.; Soltani, A.; Deng, Z.; Lam, H.T. The SPMF Open-Source Data Mining Library Version 2. In *Learning and Knowledge Discovery in Databases*; Springer International Publishing: Berlin/Heidelberg, Germany, 2016; pp. 36–40. [CrossRef]
48. Kumar, R.; Singh, K. Top-k high utility itemset mining: Current status and future directions. *Knowl. Eng. Rev.* **2024**, *39*, 1–61. [CrossRef]
49. Liu, X.; Chen, G.; Wen, S.; Zuo, W. Effective approaches for mining correlated and low-average-cost patterns. *Knowl. Based Syst.* **2024**, *302*, 112376. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.